Calcul hautes performances pour les formulations intégrales en électromagnétisme basses fréquences. Intégration, compression matricielle par ondelettes et résolution sur architecture GPGPU

Christophe Rubeck

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Génie Électrique**
Arrêté ministériel : 7 août 2006

Présentée par

Christophe RUBECK

Thèse dirigée par **Jean-Paul YONNET** et
co-encadrée par **Olivier CHADEBEC** et **Benoît DELINCHANT**

préparée au sein du **Laboratoire de Génie Électrique de Grenoble**
dans l'École Doctorale **Électronique, Électrotechnique, Automatique & Traitement du signal**

Calcul hautes performances pour les formulations intégrales en électromagnétisme basses fréquences

Intégration, compression matricielle par ondelettes et résolution sur architecture GPGPU

Thèse soutenue publiquement le **18 décembre 2012**,
devant le jury composé de :

M. Frédéric BOUILLAULT
Professeur des Universités à l'Université Paris Sud, Président
M. Mouloud FELIACHI
Professeur des Universités à l'Université de Nantes, Rapporteur
M. Christian VOLLAIRE
Professeur des Universités à l'Université Lyon 1, Rapporteur
M. Yvonnick LE MENACH
Maître de conférences à l'Université Lille 1, Membre
M. Jean-Paul YONNET
Directeur de Recherche au CNRS, Membre
M. Olivier CHADEBEC
Chargé de recherche au CNRS, Membre
M. Benoît DELINCHANT
Maître de conférences à l'Université Grenoble 1, Membre

Remerciements

Voici venu le moment pour moi de remercier toutes les personnes qui ont contribué au succès de ces trois années de thèse. Ce n'est pas évident de citer tout le monde et j'espère que les personnes que j'aurais oubliées ne m'en tiendront pas rigueur.

Je remercie tout d'abord les membres extérieurs du jury pour avoir accepté d'évaluer mon travail. Je remercie les deux rapporteurs, Mouloud FELIACHI et Christian VOLLAIRE, pour avoir estimé que mon projet de recherche est de qualité à être soutenu. Je remercie également Fréderic BOUILLAULT pour avoir présidé le jury, ainsi que Yvonnick LE MENACH.

Les personnes à qui je dois le plus sont bien entendu mes encadrants. Vous avez toujours été présents et même s'il y a eu des moments difficiles vous m'avez soutenu et le projet a pu être mené à son terme.

Je remercie chaleureusement Olivier CHADEBEC pour m'avoir aiguillé tout au long de ses travaux. Ton recul sur les méthodes intégrales et les méthodes de compactage a été d'une aide précieuse. Je te remercie également pour le temps passé à la correction du mémoire, par téléconférence entre deux parties de pêche au piranha dans les marécages infestés de crocodiles du Brésil.

Je remercie Benoît DELINCHANT pour m'avoir proposé ce sujet de thèse il y a trois ans. Tu auras noté que j'ai pris quelques libertés sur son interprétation : tu m'as lancé sur MacMMems et le calcul symbolique des intégrales et je suis parti en mode bourrin sur du calcul hautes performances par cartes graphiques. On dit qu'on ne sait jamais où une thèse

va aboutir, en voici une bonne illustration. Sinon as-tu enfin commencé à creuser ta cave à vin ? Il me tarde de venir déboucher quelques bouteilles.

Je remercie Jean-Paul YONNET, mon directeur de thèse. Tu n'as finalement pas été aussi impliqué dans mes travaux que tu le souhaitais, mais je n'oublierai pas nos séances de démontage de disques durs pour récupérer les aimants des actionneurs. Enfin démonter reste un grand mot, je doute qu'on puisse les remettre en état.

Je remercie également toutes les personnes du groupe MIPSE. En particulier Bertrand BANNWARTH, l'autre spécialiste GPGPU avec qui j'ai beaucoup échangé, Jean-Michel GUICHON, et Patrice LABIE.

Ne pouvant remercier tous les chercheurs que j'ai côtoyés au G2Elab, je vais en choisir quatre au hasard. Je commence par Jean-Louis COULOMB pour son recul scientifique incroyable dans de nombreux domaines, je continue avec Frédéric WURTZ pour son dynamisme à la tête de l'équipe MAGE, puis Laurent GERBAUD le spécialiste des randonnées de la salle numérique, enfin je conclue avec Gilles CAUFFET pour sa bonne humeur permanente.

Je remercie également le Laboratoire de Génie Électrique de Grenoble pour m'avoir hébergé ces trois années et pour m'avoir donné les moyens de réaliser ces travaux. Je remercie aussi tous les employés administratifs et techniques qui rendent la vie au laboratoire aussi fonctionnelle et agréable.

Je remercie le service informatique, en particulier Corine MARCON et Philippe BRULAT pour leur avoir fait installer une carte graphique très puissante qui ne possède pas de sortie d'affichage (logique non ?) et également Vincent DANGUILLAUME pour son aide à la mise en place du site web d'OPLAT (depuis PHP et MySQL n'ont plus de secret pour moi).

L'élément qui a rendu ces 3 années au laboratoire très agréable d'un point de vue humain est le cercle d'amis que je m'y suis fait. Vous êtes pour la plupart partis vers de nouveaux horizons mais j'espère qu'on continuera à faire des trucs ensemble de temps à autre.

Je remercie Fanny MESMIN venue de Bretagne avec qui j'ai partagé le bureau D064 pendant ces trois années. Cela a été un plaisir, les fous rires qu'on était capable de s'infliger vont me manquer, sans oublier le soutien mutuel dans les phases les plus difficiles de la thèse. Déménageuse psychopathe, tu as décidé unilatéralement d'étudier la décomposition organique de la pomme réservée pour mon goûter que tu as mise sous cloche ! Nous avons également développé une théorie intéressante sur l'extrême dangerosité des escargots sur la voie publique et de leur responsabilité présumée dans différents accidents ferroviaires inexpliqués tout en observant passer et repasser des chameaux dans le patio (j'invite le lecteur à oublier la dernière phrase qu'il vient de lire).

Je remercie Ni DING, la spécialité chinoise du laboratoire. Ton originalité et ta fantaisie ont rendu inoubliables un grand nombre de moments passés ensemble ! Je n'oublierai pas non plus ta succulente cuisine et les randonnées dans la montagne.

Je remercie encore Bertrand BANNWARTH mais pour les implications non techniques cette fois ci. Tu es le casse coup de la bande, au sens stricte. A présent quand je vais courir à la Bastille je redouble de prudence pour ne pas finir ma soirée aux urgences.

Je remercie Mickaël PETIT. Il doit te manquer une case (ça doit être l'électronique de puissance qui fait ça et/ou la réfrigération magnétique), mais je t'aime bien quand même. Je n'oublierai pas les grands dîners que tu as organisés, peu de gens sont capables de cuisiner tout seul une entrée froide, une entrée chaude, un plat, et un dessert, le tout pour 15 personnes.

Je remercie Mathieu LE-NY, l'autre spécialité bretonne. Tu es venu en montagne pour faire de la planche à voile, encore un signe qu'une thèse attaque les neurones. Heureusement que tu n'as pas choisi le surf car tu serais encore en train d'attendre la vague. Sinon pour revenir au domaine culinaire, ne te décourage pas, tu arriveras un jour à faire un gâteau au chocolat qui n'aura pas la dureté du diamant.

Je remercie Sylvain TANT pour les nombreuses soirées passées chez toi à regarder des films sur ton écran géant tout en s'explosant l'estomac de pizza. Promis, on se fera un marathon de toutes les versions longues du Seigneur des Anneaux.

Je remercie Ando Tiana RAMINOSOA, le voisin terrible. Tu m'as été d'une aide inestimable pour débusquer le fauteur de trouble qui me réveillait toutes les nuits.

Je remercie les autres de la plateforme MADEA : Manel ZIDI, Morgan ALMANZA (et sa moitié Sylvie JAUVERT qui est au laboratoire du dessus), Sylvain PEREZ, Julien ROUDAUT, Anthony FRIAS, etc.

Je remercie Wahid CHERIEF, un autre fou qui s'attaque à la réfrigération magnétique. Refais nous un grand barbecue comme cet été, c'était super !

Je remercie Roland SANFILIPPO. Ton passage au G2Elab a été court. Je te souhaite bonne chance pour la suite.

Je remercie Filipa CARDOSO qui a effectué un stage au laboratoire. Promis on ira gravir la Grande Lance de Domène !

Je remercie les gens qui sont passés par le bureau D064 et qui ont tous été d'agréable compagnie : Ardavan DARGAHI, Maria VIZITEU et Razmik DEMIRJIAN.

Je fais un clin d'œil aux disjonctés de l'électronique de puissance : Johan DELAINE, Gatien KWIMANG, Mounir MARZOUK, etc.

Dans le même ton je salue le gang des tunisiennes : Abir REZGUI, Sana GAALOUL, et Sarra BEN GDARA. Mes meilleurs vœux de bonheur aux deux dernières qui se sont mariées récemment.

Je remercie l'association des doctorants OPLAT, dont je faisais partie autrefois en tant que webmestre, pour avoir propagé une convivialité permanente au sein du laboratoire. Le lecteur notera peut être que d'une certaine façon je me remercie un peu moi-même, je tiens à le rassurer et je précise que je ne suis pas narcissique.

Je remercie vivement Pascale PHAM, ma maître de stage de fin d'études. Tu m'as donné le goût de la recherche et convaincu de poursuivre en thèse. J'espère avoir à nouveau le plaisir de travailler avec toi.

Je remercie les vieux copains : Nicklaus, Quentin, M@th!@s, Jérôme, Michael, etc. Certains ont été étonnés que je rempile pour trois ans alors qu'eux étaient si heureux d'avoir enfin terminé leurs études. Eh non je ne suis pas fou, bon alors juste un peu, pour faire plaisir.

Enfin, je remercie ma famille pour son soutient tout au long de mes études et pour avoir fait le chemin depuis la Lorraine pour assister à ma soutenance.

Table des matières

Introduction générale

Les systèmes électromagnétiques occupent une place majeure dans la société moderne. Leur conception repose essentiellement sur du prototypage virtuel, c'est la conception assistée par ordinateur. Pour répondre aux exigences des systèmes à développer, il est souvent nécessaire de tester un grand nombre de prototypes numériques pour trouver la solution technologique optimale. Il est de ce fait nécessaire de développer des méthodes et outils de modélisation performants, c'est-à-dire précis et rapides.

Les méthodes par éléments finis sont, de par leur généralité, restées longtemps dominantes dans le monde de la modélisation des dispositifs électromagnétiques basses fréquences du génie électrique. Cependant, elles souffrent parfois de quelques limitations. En effet, il est nécessaire de mailler la globalité du domaine d'étude (matériaux actifs et inactifs tel que l'air). Or dans les systèmes et microsystèmes du génie électrique, les matériaux inactifs sont souvent dominants, ce qui conduit parfois à des modèles très lourds.

Pour répondre à cet enjeu de modélisation, et donc de conception, les méthodes intégrales peuvent parfois être plus pertinentes. Contrairement aux méthodes par éléments finis, celles-ci ne nécessitent pas le maillage des matériaux inactifs, et sont donc particulièrement performantes et légères pour le calcul des interactions à distance. Les méthodes intégrales ont cependant de lourds inconvénients. En effet, ceux sont des méthodes à interactions totales qui mènent à la construction de système d'équations linéaires dits pleins, qui sont donc très coûteuses en temps de calcul et en espace de stockage mémoire (complexité N^2 avec N le nombre de degrés de liberté). La résolution du système d'équations par une méthode itérative est également de complexité parabolique.

Ces travaux de thèse se placent dans un contexte de calcul hautes performances (HPC). Nous étudierons les bénéfices que peut apporter le parallélisme, c'est-à-dire l'utilisation de plusieurs processeurs, pour accélérer les calculs nécessaires à la mise en œuvre des méthodes intégrales. Le cœur des travaux portera sur l'exploitation des cartes graphiques (GPGPU) qui, de par leur excellent rapport puissance sur coût, sont devenues très intéressantes pour effectuer des calculs scientifiques massivement parallèles.

Le coût en temps de calcul de la complexité parabolique des méthodes intégrales est minimisé grâce à cette architecture particulière. Cependant le stockage de la matrice du système d'équations linéaires reste une difficulté importante. Par exemple avec 4 Go de mémoire les problèmes sont limités à 20.000 degrés de liberté en double précision alors que l'ordre de grandeur d'un problème de complexité industrielle peut se situer parfois autour de 100.000. Nous appliquerons une méthode de compression matricielle par ondelettes, proche des algorithmes utilisés en compression d'image. Cette méthode a l'avantage d'être peu invasive, c'est-à-dire que les méthodes de construction de la matrice ne sont pas modifiées. Une conséquence est que la complexité de calcul reste parabolique. Nous utilisons alors à nouveau le parallélisme des processeurs graphiques pour accélérer le calcul. Notons que la compression par ondelettes est une méthode à perte d'information. Nous développerons alors un critère pour contrôler l'erreur introduite par la compression.

Nous présentons au Chapitre I une introduction au calcul hautes performances. Nous débutons par une présentation rapide des différentes architectures informatiques et des optimisations des algorithmes qui en découlent. Nous aborderons ensuite le calcul parallèle. Nous porterons une attention particulière au parallélisme sur architecture GPGPU. L'enjeux est de partitionner un algorithme en plusieurs tâches et de les distribuer soit sur un ordinateur qui possède plusieurs processeurs, soit sur un réseau d'ordinateurs, soit encore sur un calculateur graphique qui possède des centaines de processeurs. Pour chaque architecture parallèle des stratégies différentes doivent être adoptées, notamment sur la gestion et le partage de la mémoire ainsi que sur la synchronisation des tâches.

Nous présenterons ensuite, dans le Chapitre II, une formulation intégrale en potentiel électrostatique performante et massivement parallélisable dans le but d'être portée sur architecture GPGPU. Cette formulation sera tout d'abord vectorisée afin d'accélérer les calculs par l'exploitation du « pipelining » des processeurs. L'intégration numérique par

points de Gauss est appréciée pour sa rapidité mais elle peut parfois se montrer imprécise voire singulière. L'intégration analytique est quant à elle appréciée pour sa précision mais elle est coûteuse en temps de calcul. L'intégration par points de Gauss et l'intégration analytique seront donc hybridées afin d'obtenir le meilleur ratio entre précision et temps de calcul.

Dans le Chapitre III, nous réduisons le coût calculatoire en temps de notre formulation intégrale grâce au parallélisme. Les architectures parallèles exploitées sont : le PC multicoeur ; le cluster (ensemble de PCs reliés en réseau) ; ainsi que le GPGPU. Chacune de ces architectures possède des spécificités qui influeront sur la manière de paralléliser les calculs, les points les plus critiques étant la gestion des mémoires et des communications. L'enjeu de ces travaux est d'adapter les codes sur chacun de ces modèles de programmation puis de les optimiser. Plusieurs stratégies seront confrontées pour le calcul du système d'équations linéaires et de sa résolution.

Dans le Chapitre IV, nous réduisons le coût du stockage mémoire du système d'équations. Nous tentons tout d'abord un portage des méthodes multipolaires rapides (FMM) sur GPGPU, mais cet algorithme est très séquentiel et sa parallélisation est décevante. Nous proposons ensuite de réduire les besoins en mémoire en compressant la matrice d'interaction par ondelettes. Cette méthode a l'avantage d'être peu invasive, cependant elle reste de complexité parabolique car le calcul de l'ensemble de la matrice est nécessaire. Nous utiliserons alors le parallélisme sur GPGPU pour accélérer ces calculs.

Enfin, nous appliquons dans le Chapitre V notre méthodologie sur un cas de complexité industrielle : le calcul des capacités parasites d'un variateur de vitesse.

Nous concluons le mémoire et proposons quelques perspectives.

Chapitre I

Introduction au calcul hautes performances

Sommaire

Résumé

Ce chapitre présente une introduction au calcul hautes performances. Il débute par les présentations des différentes architectures informatiques et de leurs optimisations. Le calcul parallèle est ensuite présenté, plus particulièrement le calcul sur cartes graphiques. L'enjeu ici est de partitionner un algorithme pour le paralléliser, soit sur un ordinateur qui possède plusieurs processeurs, soit sur un réseau d'ordinateurs, soit encore sur un calculateur graphique qui possède des centaines de processeurs. Pour chaque type de parallélisme, des stratégies différentes doivent être adoptées, notamment sur la gestion et le partage de la mémoire ainsi que sur la synchronisation des tâches.

1 Introduction

La modélisation numérique est devenue incontournable dans le monde de la conception industrielle et de la recherche scientifique. Les problèmes à résoudre sont de plus en plus complexes. Pour y répondre efficacement, il est nécessaire d'adopter une approche de calcul hautes performances.

Dans le domaine de la simulation numérique, la performance est définie par le rapport de la précision sur le temps de calcul. Par conséquent, pour augmenter les performances d'un calcul, il doit devenir soit plus précis à temps constant, soit plus rapide à précision constante, que la version initiale. La précision est généralement donnée par le modèle numérique choisi (maillage, formulations physiques, etc.). Ainsi ce travail se concentrera sur l'amélioration des temps de calcul.

Comment accélérer un calcul numérique ? Une première solution est d'augmenter la fréquence d'horloge du processeur. Il s'agit d'un remplacement coûteux du matériel ou d'overcloking[1], cette solution ne sera pas considérée dans ces travaux. De plus, la montée en fréquence des processeurs semble avoir atteint ses limites. Une seconde piste d'investigation est de permettre l'exécution simultanée de plusieurs instructions. Se trouvent ici le calcul parallèle, qui consiste à utiliser plusieurs composants électroniques (processeur multicœur, multi-CPU, GPGPU[2], PCs en réseaux, etc.), et le pipelining qui est un parallélisme d'instruction au sein du même processeur. Une dernière méthode pour accélérer un calcul numérique est l'amélioration des accès mémoires. En effet, les transferts de données sont souvent le goulot d'étranglement. Une attention particulière sera mise sur la gestion de la mémoire, surtout lors de l'utilisation des cartes graphiques.

Tout d'abord, des généralités sur les architectures informatiques sont présentées dans ce chapitre. En effet, comprendre le fonctionnement d'un ordinateur permet déjà des optimisations simples à mettre en œuvre telle que la vectorisation. Ensuite, le calcul

[1] Overcloking : manipulation consistant à augmenter la fréquence d'horloge du processeur au delà des spécifications du constructeur. Il en résulte un accroissement de la puissance de calcul.

[2] General-Purpose computation on Graphics Processing Units : utilisation de cartes graphiques pour effectuer des calculs en supplément du processeur.

parallèle qui est la problématique principale de ce mémoire est introduit. Les différents types de parallélisme sont décrits, chacun ayant ses particularités qui définiront des stratégies de parallélisation qui lui seront propres. Les architectures étudiées sont des PC à processeurs multicoeur, des réseaux de PCs et enfin des cartes graphiques, programmables depuis 2007, qui possèdent des centaines de cœurs. Les enjeux sont ici de partitionner les problèmes sur plusieurs tâches, gérer les mémoires et synchroniser les tâches.

2 Généralités sur les architectures informatiques

Pour bien comprendre les enjeux du calcul hautes performances, il est utile d'avoir une idée précise du fonctionnement d'un ordinateur.

2.1 Machine de Von Neumannn

Les architectures des PCs sont des dérivées de la machine de Von Neumannn [Neumann 47, Bersini 08, Patterson 03]. Cette architecture est nommée d'après le mathématicien John von Neumannn. Son originalité réside dans l'utilisation d'une mémoire unique pour le stockage à la fois des instructions et des données nécessaires aux calculs (Figure 1). Cette architecture se décompose en quatre parties :

1. L'unité arithmétique et logique (ALU) qui effectue les opérations de calcul et de comparaison.
2. L'unité de contrôle chargée du séquençage des opérations.
3. La mémoire, qui se décompose en deux parties : la mémoire vive (RAM[3], cache des processeurs) qui contient les programmes en cours d'exécution et les données volatiles, et la mémoire permanente (disque dur, clef USB, etc.) qui contient les programmes et les données de bases.
4. Les entrées-sorties qui permettent la communication avec l'extérieur (clavier, écran, réseau, etc.).

[3] Random Access Memory, il s'agit de la mémoire vive d'un ordinateur.

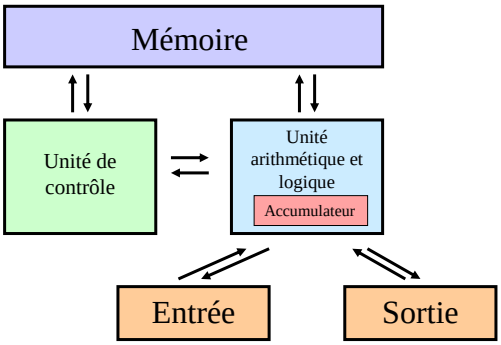


Figure 1 : Machine de Von Neumann.

2.2 Hiérarchisation des mémoires

Les architectures actuelles (Figure 2) héritent de la machine de Von Neumannn. S'y retrouvent les entrée/sorties, les mémoires volatiles et permanentes, et les unités de calcul. Cependant, les mémoires volatiles se divisent en plusieurs catégories : la RAM et les caches des processeurs. Les caches permettent un accès plus rapide à la RAM en copiant certaines données qui sont souvent appelées.

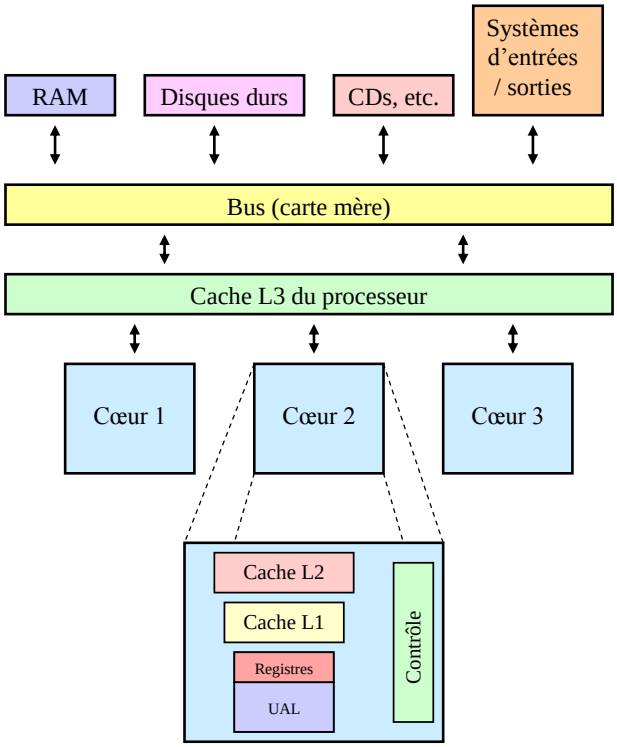


Figure 2 : Architecture moderne avec CPU multicoeur à 3 niveaux de mémoire cache.

La RAM et la cache sont toutes deux à base de semi-conducteurs. La RAM est construite sur de la DRAM[4], chaque bit est stocké dans un pico condensateur qui doit être rafraîchi périodiquement pour éviter les fuites, c'est une mémoire dynamique. Cette mémoire est généralement capable de stocker plusieurs gigaoctet de données. Les mémoires caches et les registres sont construits sur de la SRAM[5]. Contrairement à la DRAM, elle n'a pas besoin de rafraîchissement, cependant elle n'est disponible qu'en faible quantité et pour un coût très supérieur à la DRAM. C'est pourquoi les mémoires sont hiérarchisées (Figure 3) en fonction de leur capacité de stockage et de leur vitesse (liée à leur coût).

Les capacités présentes dans les caches des processeurs sont de l'ordre du mégaoctet pour la L3 [Intel 11b], de la centaine de Ko pour la L2, la dizaine de Ko pour la L1 et du Ko pour les registres. Les vitesses de transfert sont de l'ordre du Go/s entre la RAM et le cache L3, et de la dizaine de Go/s entre les caches (ainsi que pour les registres).

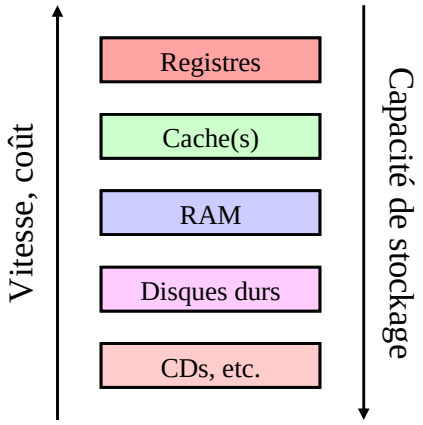


Figure 3 : organisation hiérarchique des mémoires.

Par exemple, le processeur Intel Core i7 2600 [CPU World] possède 4 cœurs de calcul, 8 Mo de mémoire cache L3, 256 ko de L2 par cœur et 32 ko de L1 par cœur.

2.3 Performance de calcul

La performance des ordinateurs est limitée par la latence (temps pour un seul accès) et la bande passante (nombre d'accès par unité de temps) de la mémoire. Le temps de latence

[4] Dynamic Random Access Memory.

[5] Static Random Access Memory.

est de plusieurs ordres de grandeur supérieur au temps d'un cycle CPU. Les accès mémoires constituent bien un goulot d'étranglement qu'il est nécessaire d'optimiser.

Le niveau sur lequel il est possible d'agir le plus efficacement est le transfert de données entre la mémoire vive et le processeur. Il est important que l'accès aux données soit coalescent, c'est-à-dire d'appeler des blocks de mémoire consécutifs. La Figure 4 montre un accès coalescent. Un flux de données est créé entre la mémoire et le processeur, ainsi les temps de latence sont diminués, et par conséquent la bande passante est maximale.

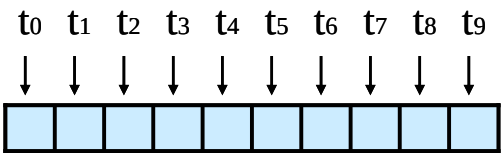


Figure 4 : Accès coalescent à la mémoire.

La Figure 5 montre un accès non séquentiel à t=2 et à t=3, ainsi qu'une discontinuité à t=6 et à t=7. Ces accès sont bien entendu non coalescents. La perte de performances d'un accès non coalescent peut se traduire par un ralentissement du programme de l'ordre de 50 fois (test effectué sur un petit programme de multiplication matricielle en C).

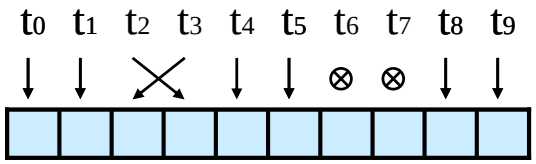


Figure 5 : Accès non coalescent, accès non séquentiel à t_2 et t_3, discontinu à t_6 et t_7.

Il existe un parallélisme d'instruction au sein du processeur, c'est le pipelining. En effet, les processeurs modernes permettent d'exécuter plusieurs instructions simultanément. Soit un processeur qui accompli une opération en 5 étapes [Bersini 08] :

1. IF (Instruction Fetch) charge l'instruction à exécuter dans le pipeline.
2. ID (Instruction Decode) décode l'instruction et adresse les registres.
3. EX (Execute) exécute l'instruction.
4. MEM (Memory) dénote un transfert entre un registre vers la mémoire.
5. WB (Write Back) stocke le résultat dans un registre.

Dans le cas où chaque étape met 1 cycle d'horloge pour s'exécuter, il faut alors 5 cycles pour exécuter une instruction et donc 15 pour 3 instructions (Figure 6).

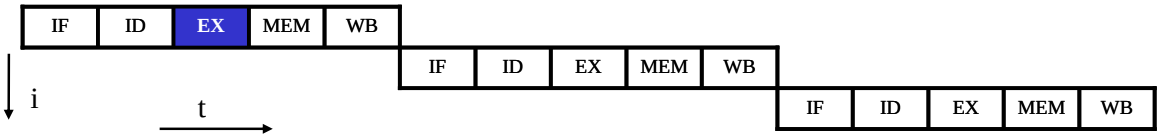


Figure 6 : Exécution de 3 instructions sur un processeur sans pipeline. 15 cycles d'horloge sont nécessaires.

Soit maintenant un processeur possédant 5 pipelines (Figure 7). Chaque instruction se déroule toujours en 5 étapes, mais l'ensemble des 5 instructions se déroule en 9 cycles au lieu de 25. À t=5 tous les pipelines sont sollicités. Ce parallélisme est bien adapté au traitement de grands volumes de donnés (cf. vectorisation).

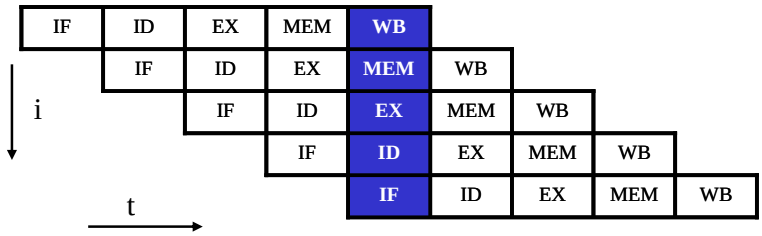


Figure 7 : Exécution de 5 instructions sur un processeur possédant 5 pipelines. Seuls 9 cycles d'horloges sont nécessaires.

Toutes les opérations arithmétiques n'ont pas le même coût. Flynn [Flynn 95] propose une estimation du coût en cycles d'horloge des opérations arithmétiques sur la base qu'une addition coûterait 1 cycle d'horloge en temps de calcul (Figure 8). Le coût d'une division ou d'un modulo est très important. Il est donc indispensable de les utiliser le moins possible.

Opération		Simple précision	Double précision
Nombre entier	Addition, Soustraction	1	1
	Multiplication	2	3
	Division, Modulo	16	32
Nombre à virgule flottante	Addition, Soustraction	2	2
	Multiplication	3	4
	Division, Modulo	13	27

Figure 8 : Coût des opérations arithmétiques en cycles d'horloge.

2.4 Optimisation des codes de calcul

Sont présentées ici quelques techniques d'optimisation des codes de calcul numérique [Dowd 98]. Il est important, une fois un code écrit et validé, d'optimiser sa vitesse d'exécution. Pour se faire, il faut essentiellement tenir compte du fonctionnement de l'ordinateur comme présenté précédemment. D'autres exemples sont proposés dans l'Annexe A.

2.4.1 Notion de vectorisation

La vectorisation consiste à traiter les données continûment par paquets, c'est-à-dire de travailler sur des tableaux et non plus des variables. Cette technique permet d'exploiter les deux notions vues précédemment qui sont la coalescence et le pipelining [Golub 83]. L'opération vectorielle est décodée une seule fois par le processeur, et les constantes sont placées dans sa mémoire cache. Les éléments des vecteurs sont ensuite soumis un à un au processeur. Ce transfert de données entre la RAM et le processeur est coalescent, donc très performant. De plus, les pipelines du processeur sont pleinement alimentés ce qui accroît la vitesse de calcul.

En modélisation numérique, de très grands nombres de données sont couramment traitées, la vectorisation est alors nécessaire afin d'obtenir de bonnes performances. La vectorisation s'effectue souvent par le biais de bibliothèques spécialisées telles que BLAS par exemple [BLAS]. BLAS est une bibliothèque d'opérations vectorielles et matricielles. Elle est décomposée en trois sections, BLAS 1 pour les opérations vecteur/vecteur (y=alpha*x + y), BLAS 2 pour les opérations matrice/vecteur (y=alpha*A*x + y) et BLAS 3 pour les opérations matrice/matrice (C=alpha*AB+C).

2.4.2 Stockage des matrices

Dans certaines bibliothèques de manipulation de matrices disponibles sur Internet, les matrices sont définies ainsi : M[i][j], où i et j sont les index de l'élément de la matrice. Il peut s'agir d'un tableau de tableaux, l'accès aux données n'est alors pas toujours performant.

Le standard est de stocker les matrices dans des tableaux, soit en format ligne, soit en format colonne [BLAS]. Le Fortan utilise le format colonne, c'est-à-dire que les matrices

sont définies de la façon suivante : M(i,j) = T[i +j*dj], avec T le tableau dans lequel les éléments de la matrice sont stockés et dj l'incrément lors du déplacement selon la direction j. Ici dj prend pour valeur le nombre de ligne de la matrice. Le C/C++ ainsi que le Java utilisent le format ligne. Les matrices sont alors définies ainsi : M(i,j) = T[i*di+j], avec di l'incrément lors du déplacement selon la direction i qui prend pour valeur le nombre de colonnes de la matrice.

Il est également possible de définir une matrice de la façon suivante : M(i,j) = T[i*di+j*dj+o], avec o un décalage par rapport au début du tableau. Si la matrice est déroulée dans le sens des lignes par exemple, alors di sera égal au nombre de colonnes et dj prendra 1 comme valeur. L'avantage de cette définition est que l'accès à la transposée se fait par une simple inversion des indices di et dj ! De plus l'index de décalage permet de déplacer facilement dans le tableau et de définir simplement des sous-matrices.

2.4.3 Division par une constante

Soit la boucle suivante :

```
for (i=0 ; i<n ; i++) {
    A[i] = A[i] / sqrt(x*x+y*y)
}
```

Cette boucle contient une division par une constante, hors les divisions sont très coûteuses. De plus, le terme au dénominateur est également coûteux à calculer, il doit être calculé hors de la boucle :

```
cste = 1 / sqrt(x*x+y*y)
for (i=0 ; i<n ; i++) {
    A[i] = A[i] * cste
}
```

Le code a été accéléré en théorie d'un facteur de 8 à 10 uniquement grâce à la suppression de la division. L'accélération est plus importante encore due au fait que la fonction racine carrée n'est plus appelée à chaque itération. Cet exemple particulièrement pathologique montre qu'il n'est pas nécessaire de complexifier énormément l'algorithme pour être plus performant.

2.4.4 Boucles imbriquées

Soit l'algorithme très connu de multiplication matricielle. Les matrices sont ici stockées au format ligne par exemple. Classiquement, il s'écrit (pour une matrice n x n) :

```
for (i=0 ; i<n ; i++) {
    for (j=0 ; j<n ; j++) {
        sum = 0
        for (k=0 ; k<n ; k++)
            sum = sum + A[i,k] * B[k,j]
        }
        C[i,j] = sum
    }
}
```

Ici, l'accès à B[k,j] n'est pas coalescent ! La perte du pipelining peut provoquer un ralentissement d'un ou deux ordres de grandeur ! L'algorithme optimisé peut être réécrit de cette façon :

```
for (i=0 ; i<n ; i++) {
    for (j=0 ; j<n ; j++) {
        C[i,j] = 0
    }
}

for (k=0 ; k<n ; k++) {
    for (i=0 ; i<n ; i++) {
        scale = A[i,k]
        for (j=0 ; j<n ; j++)
            C[i,j] = C[i,j] + B[k,j] * scale
        }
    }
}
```

L'accès coalescent à B[k,j] est retrouvé pour le prix d'une variable supplémentaire dans l'algorithme. En réalité ce prix est quasi nul car si le compilateur fait bien son travail elle sera placée en cache au plus près du processeur.

Remarque : si les matrices étaient stockées au format colonne, ce serait l'accès à A[i,k] qui aurait posé un problème de coalescence.

3 Introduction au calcul parallèle

Le terme de calcul parallèle se réfère ici à l'utilisation de plusieurs processeurs, (multicœur, multi-CPU, réseau) pour faire des calculs. Il y a plusieurs intérêts à faire du calcul parallèle, tout d'abord être plus rapide (multicoeur en particulier), ou encore résoudre des problèmes plus importants grâce au cumul des ressources comme la mémoire vive qui est souvent un facteur limitant. Une dernière motivation est tout simplement la possibilité d'effectuer plusieurs tâches en même temps.

3.1 Fin de règne du calcul séquentiel

Un programme séquentiel est un programme qui ne fait qu'une opération à la fois, contrairement à un programme parallèle qui peut traiter plusieurs opérations simultanément.

Gordon Earle Moore publia en 1965 une loi dans *Electronics magazine* qui porte son nom. La loi de Moore est une loi empirique qui prédit que le nombre de transistors dans les processeurs double tous les 18 mois [Schaller 97]. Cette loi s'est révélée étonnamment exacte, et devrait le rester jusqu'en 2015. Les processeurs possèderont alors plus de 15 milliards de transistors. Ensuite des limites physiques apparaîtront (effet tunnel, désintégration bêta) et il sera impossible de graver des transistors plus finement (avec les matériaux actuels).

La loi de Moore semble avoir déjà atteint ses limites d'un point de vue industriel. En effet, graver toujours plus finement n'est plus forcément rentable, et il se pose également des problèmes de dissipation thermique [Kish 02]. La solution trouvée par les fondeurs est de multiplier les cœurs de calcul. Dorénavant, la plupart des processeurs vendus dans le commerce sont multicoeurs.

Finalement, la loi de Moore reste exacte grâce à la multiplication des cœurs, à condition néanmoins que les programmes sachent les exploiter.

3.2 Terminologie du parallélisme

Sont présentées ici quelques définitions de termes qui sont couramment rencontrés en calcul hautes performances.

Tâche et tâche parallèle

Une tâche est une portion d'un travail à effectuer par un ordinateur. Il s'agit typiquement d'un ensemble d'instructions exécutées sur un processeur. Une tâche parallèle est une tâche qui peut s'exécuter sur plusieurs processeurs.

Processus et thread[6]

Un processus est un programme en cours d'exécution, il est créé par le système d'exploitation. Il contient un ou plusieurs threads. Un thread est constitué d'une ou plusieurs tâches. Dans une programmation multi-thread, chaque thread possède une mémoire propre et partage la mémoire globale du thread parent.

Mémoire partagée

La mémoire partagée est une mémoire accessible par toutes les tâches. C'est l'architecture mémoire utilisée dans la programmation multithread, chaque thread partageant la mémoire globale du thread parent. Ce modèle est simple à mettre en œuvre sur une machine multicoeur qui a pour particularité que toutes les unités de calcul ont accès à la même mémoire RAM.

Mémoire distribuée

Il s'agit ici d'une mémoire non commune aux différentes tâches, car elle est physiquement répartie sur plusieurs machines. Elle est accessible via des communications entre les différentes tâches (par fichiers, réseau, etc.).

[6] Thread : « ligne d'exécution »

Accélération

L'accélération est le rapport des temps d'exécution entre le programme séquentiel et sa version parallèle. Une accélération idéale est le nombre de processeurs utilisés (voir scalabilité).

Massivement parallèle

Ce terme est utilisé pour désigner des architectures matérielles qui contiennent un grand nombre de processeurs. Ce nombre est évidement en constante augmentation, à l'heure de la rédaction de ce manuscrit il se situe entre cent et mille.

Communications

Les communications sont des échanges des données entre plusieurs tâches parallèles. Ces communications sont généralement faites à travers un réseau.

Synchronisation

Une synchronisation est un point d'arrêt dans une tâche qui se débloque lorsque les autres tâches sont arrivées au même niveau. Le but est généralement de coordonner les tâches vis-à-vis des communications.

Granularité

La granularité est le rapport entre les temps de calcul et les temps de communication. Le grain est dit grain grossier si une grande quantité de calcul est traitée entre les communications et de grain fin si au contraire il y a peu de calcul entre les communications.

Scalabilité

La scalabilité d'un système parallélisé et sa capacité à fournir une accélération proportionnelle au nombre de processeurs. Elle dépend du matériel (vitesse des bus, etc.), de la capacité de l'algorithme à être parallélisé et également de la programmation. Une scalabilité idéale est 1.

3.3 Taxinomie de Flynn

La classification des architectures parallèles la plus utilisée est la classification de Flynn (Figure 9). Elle a été proposée par Michael J. Flynn en 1966 [Flynn 72, Duncan 90, Kumar 94].

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Figure 9 : Classification de Flynn.

Les quatre catégories définies par Flynn sont classées selon le type d'organisation du flux de données et du flux d'instructions :

- **SISD** : il s'agit d'une architecture séquentielle qui contient un seul flot d'instruction et un seul flot de donnés.
- **SIMD** : ici toutes les unités de traitement exécutent le même calcul sur des données différentes pour produire des résultats différents.
- **MISD** : un seul flot de données alimente plusieurs unités de traitement. Cette architecture est très peu utilisée.
- **MIMD** : chaque unité de traitement peut gérer un flot d'instructions différent, sur des données différentes. C'est l'architecture la plus courante, elle est notamment rencontrée sur les PCs.

3.4 Architectures parallèles

La programmation parallèle est fortement liée à l'architecture parallèle utilisée. Sont présentés ici les architectures parallèles les plus courantes.

Multicœurs

Les processeurs dits multicœurs sont des processeurs qui contiennent plusieurs unités de calcul. En 2012, il s'agit de l'architecture la plus courante, elle représente le nouveau standard des PCs. Le nombre de cœurs de calcul se situe couramment entre 2 et 8. C'est une architecture à mémoire partagée car tous les cœurs ont accès à la mémoire vive. Pour exploiter cette architecture, il est courant de diviser le problème en *n* threads, avec *n* le nombre de cœurs de la machine. Ce partitionnement est assez facile à mettre en œuvre car le nombre de partitions est faible, et de plus, la mémoire est partagée entre les threads.

Architecture SMT

L'architecture SMT pour Simultaneous Multithreading permet d'optimiser l'utilisation des ressources des processeurs multicoeurs, comme par exemple remplir les cycles perdus (Figure 10). Les pipelines sont partagés entre plusieurs threads, les registres et les caches également. Les performances individuelles de chaque thread sont dégradées, mais l'exécution de l'ensemble des threads est améliorée [Tullsen 95, Lo 97].

Par exemple, l'Intel Core i7 2600 citée précédemment, est une architecture SMT [Intel 11]. En effet, le processeur possède 8 cœurs logiques alors qu'il en existe physiquement que 4. Ce mécanisme qui permet de simuler la présence de plus de cœurs qu'il n'en existe physiquement est appelé hyperthreading par Intel.

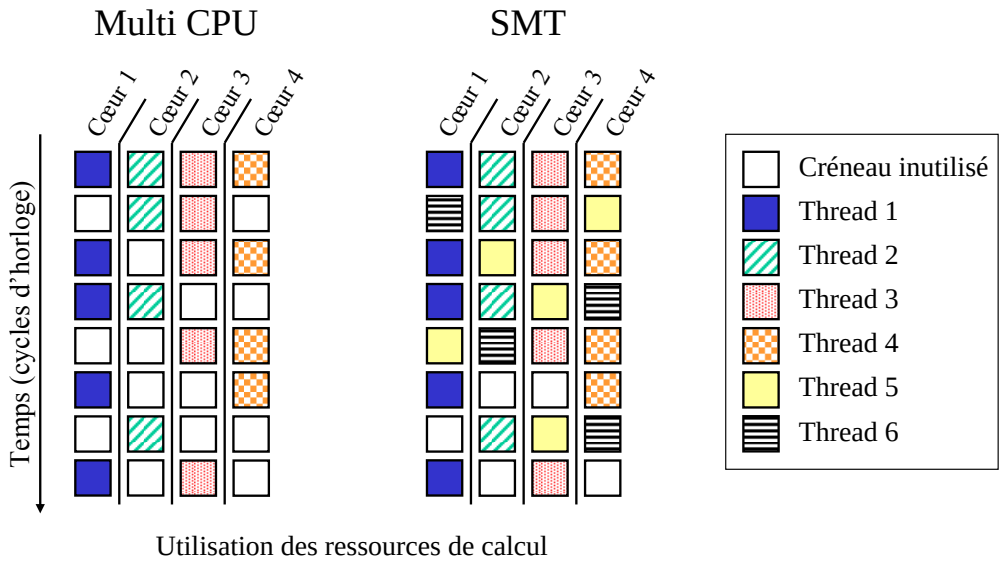


Figure 10 : L'architecture SMT permet d'optimiser l'exploitation des ressources de calcul des processeurs en saturants leurs pipelines.

Cluster de calcul

Un cluster de calcul, ou ferme de calcul en français, désigne un ensemble d'ordinateurs reliés en réseau. L'échelle de ce réseau est locale : une armoire, une pièce, ou un bâtiment. Il s'agit d'une architecture à mémoire distribuée, chaque machine n'a pas accès à la mémoire des autres. Les communications se font via un réseau. Pour le développeur de code de calcul parallèle, il lui faudra également gérer les temps de communication réseau qui peuvent devenir un goulot d'étranglement (cf. la granularité).

Ce parc de machine est généralement géré par un serveur qui fait office de gestionnaire de tâches. C'est lui qui va distribuer les tâches sur les machines, gérer les ressources disponibles, ainsi que les pannes des machines.

Grille de calcul

Une grille de calcul est un cluster de clusters. L'échelle est beaucoup plus importante qu'un cluster, elle peut être d'une ville, d'une région, d'un pays voire de toute la planète avec pour exemple le projet BOINC[7] [BOINC]. Plus localement, le projet CIMENT[8] [CIMENT] dans la région grenobloise peut être cité. La Figure 11 présente la grille de calcul CIMENT. Cet exemple est intéressant car le parc de machines est hétérogène, il est constitué de serveurs de calcul x86, Itanium, et même de PCs standards issus d'une salle de travaux pratiques.

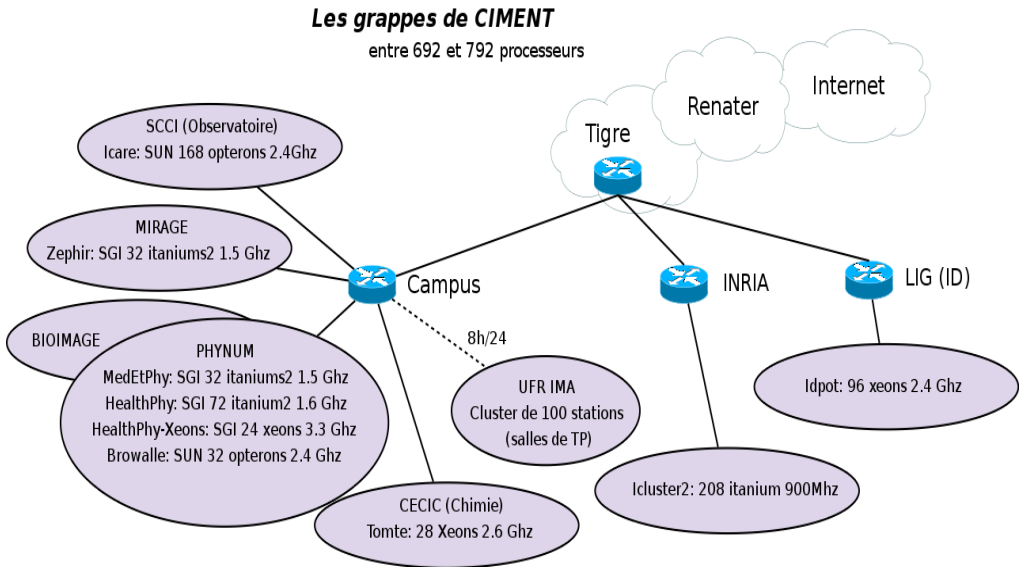


Figure 11 : Diagramme de la grille de calcul CIMENT. Le parc de machines est très hétérogène et les niveaux de réseaux sont multiples. Source : présentation CIMENT, Bruno Bzeznik et Laurent Desbat.

GPGPU

Le GPGPU pour General-Purpose computation on Graphics Processing Units désigne l'utilisation de cartes graphiques, initialement dédiées à l'affichage, en renfort des

[7] Berkeley Open Infrastructure for Network Computing : c'est un projet de calcul bénévole consacré à une grande variété de domaines de recherche, tel le séquençage de l'ADN, la recherche de pulsars, etc.

[8] Calcul Intensif / Modélisation / Expérimentation Numérique et Technologique. CIMENT vise au développement de projets de calcul de taille mésoscopique au sein des Universités Grenobloises.

processeurs conventionnels pour effectuer des calculs. Le point fort de cette architecture est sa très grande puissance de calcul brute qui est de l'ordre de celle d'un petit cluster de calcul. L'avantage par rapport à un cluster est l'absence de gestion d'un réseau et d'un parc de PCs, le tout pour un coût moindre que celui d'un cluster ! Cependant il s'agit d'une architecture massivement parallèle et tous les algorithmes ne s'y prêtent pas.

3.5 Limites et coût du parallélisme

Idéalement, l'accélération d'un programme parallélisé devrait être le nombre de cœurs de calcul utilisé (scalabilité optimale). La réalité n'est malheureusement pas aussi simple car il existe toujours des parties séquentielles qui peuvent représenter un coût non négligeable. Amdhal proposa en 1967 [Amdahl 67] le modèle suivant pour décrire l'accélération que peut apporter une machine à n processeurs (1) :

$$Acc(n) = \frac{1}{r_S + \frac{r_P}{n}} \tag{1}$$

Où $r_s + r_p = 1$, et r_s et r_p sont respectivement les proportions séquentielles et parallélisable du programme. Ce modèle limite l'accélération que peut apporter la parallélisation. En effet, la limite quand n tend vers l'infini est finie (Figure 12).

Gustafson proposa en 1988 [Gustafson 88] une réévaluation de la loi de Amdhal car il jugeait cette dernière trop pessimiste (Figure 13). La parallélisation ne sert pas uniquement à réduire le temps de calcul d'un problème donné, comme Amdhal le considère, mais surtout à traiter des problèmes plus importants. Dans ce cas, la proportion séquentielle reste plus ou moins constante. En effet, dans nos problèmes, la partie séquentielle contient typiquement l'extraction d'un maillage. Cette dernière est de complexité N avec N le nombre de nœuds par exemple. Le problème physique à résoudre peut être quant à lui de complexité N log N s'il s'agit d'éléments finis, ou N² s'il s'agit d'une méthode intégrale par exemple. L'augmentation du coût des opérations séquentielles est minime par rapport à celles parallélisables. Par conséquent Gustafson définit une loi d'accélération linéaire (2) :

$$Acc(n) = r_S + n.r_P \tag{2}$$

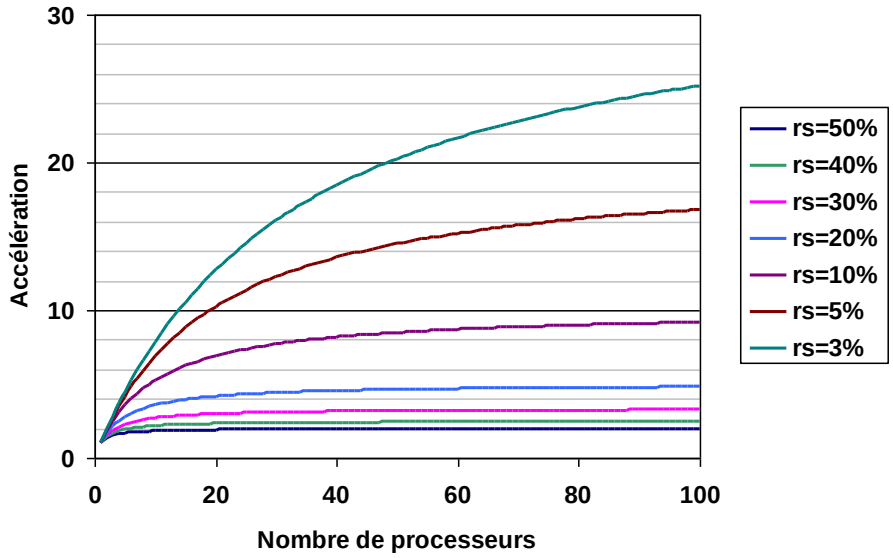


Figure 12 : Loi de Amdhal, accélération en fonction du nombre de processeurs pour différentes proportion de programme séquentiel. L'accélération est ici limitée.

La vérité se situant probablement entre les lois de Amdhal et de Gustafson, Ni proposa une loi plus fine [Ni 90] qui tient également en compte des temps de communication entre les processeurs, cette loi ne sera pas présentée dans ce manuscrit.

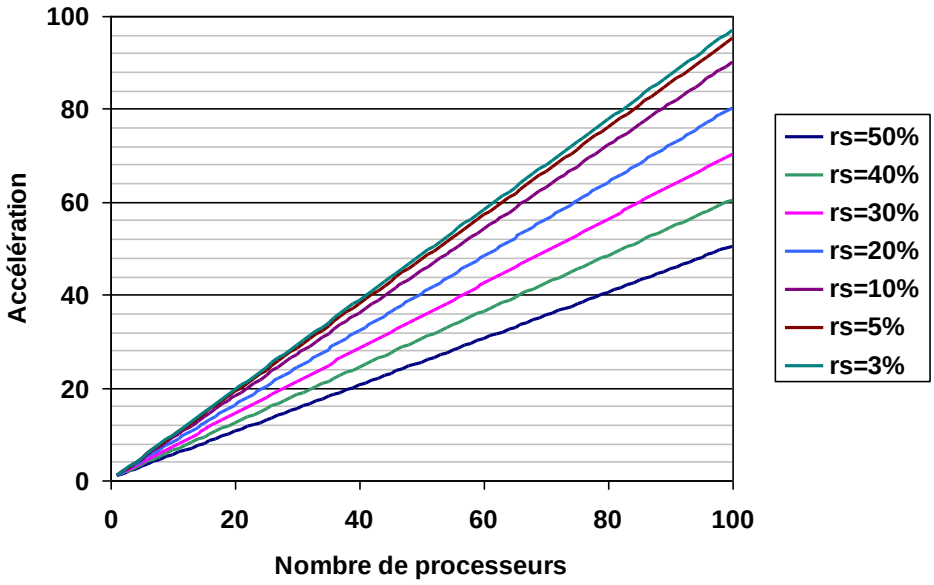


Figure 13 : Loi de Gustafson, accélération en fonction du nombre de processeurs pour différentes proportion de programme séquentiel. L'accélération suit une loi linéaire.

4 Enjeux de la programmation parallèle

La programmation parallèle nécessite de décoder les relations entre les architectures et les applications. L'architecture doit être choisie en fonction des besoins (puissance de calcul, quantité de mémoire, architecture mémoire, etc.). Il est nécessaire de comprendre le comportement d'un algorithme afin de l'adapter à l'architecture choisie. Les principaux enjeux de la programmation parallèle sont discutés dans cette section.

4.1 Partitionnement du problème

La première difficulté dans la transformation d'un algorithme séquentiel en un algorithme parallèle est la division du problème en portions pouvant être traitées simultanément. Cette opération est appelée partitionnement. Cette phase d'analyse du problème doit être réalisée au mieux car c'est elle qui mènera à une bonne scalabilité. Le choix d'un partitionnement est fortement corrélé à l'architecture utilisée.

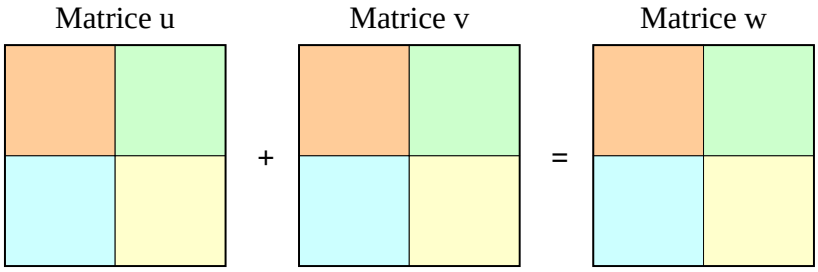


Figure 14 : Partitionnement d'une addition matricielle en 4 tâches.

Dans l'exemple présenté à la Figure 14, une addition matricielle peut être facilement partitionnée en 4 tâches. Chaque tâche, totalement indépendante des autres, effectue l'addition d'une sous matrice, la scalabilité est maximale dans cet exemple très simple.

Le partitionnement d'une multiplication matricielle est un peu plus sophistiqué (Figure 15). Les blocs des matrices u et v sont lus plusieurs fois simultanément par les différentes tâches. Il peut en résulter une baisse des performances, mais qui reste limitée car la lecture est une opération très rapide.

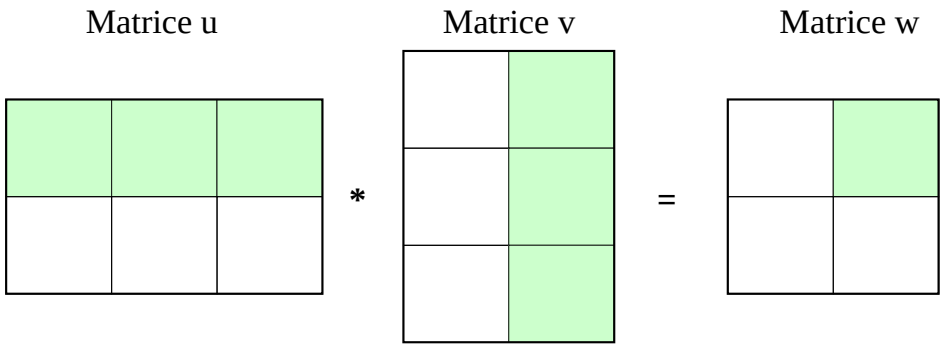


Figure 15 : Partitionnement d'une multiplication matricielle en 4 tâches.

4.2 Gestion des tâches

Une conséquence directe du partitionnement est la distribution des calculs sur les différentes tâches. La Figure 16 présente une distribution déséquilibrée de l'occupation des tâches au cours du temps. La scalabilité est ici fortement réduite compte tenu que certaines tâches sont peu actives. Il est nécessaire d'équilibrer les tâches afin de minimiser les temps morts. Chaque tâche doit donc recevoir une quantité de travail équivalente. Dans le cas d'opérations vectorielles ou matricielles, le travail étant parfaitement identique sur chaque élément du vecteur ou de la matrice, la solution est de découper le travail en parts égales sur chaque tâche. Dans les cas plus complexe il est nécessaire d'analyser les performances des codes de calculs avec des outils spécialisés.

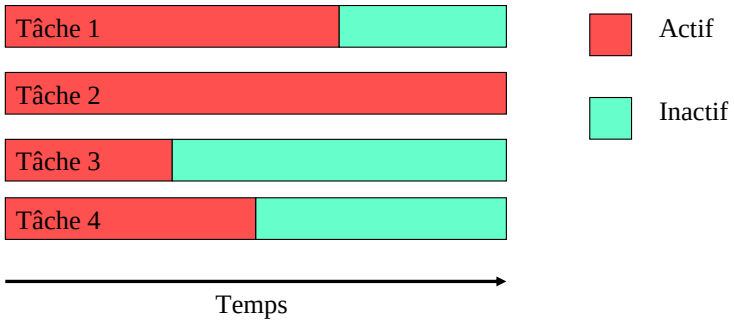


Figure 16 : Occupation des tâches en fonction du temps. La répartition est déséquilibrée ici.

4.3 Communications entre les tâches

Le besoin de communications entre les tâches dépend du problème. Une tâche au sein d'une addition matricielle par exemple n'a pas besoin des informations que peuvent contenir les autres tâches, il n'y aura pas de communication. Dans le cas d'un problème par différences finies par exemple, les éléments ont besoin de connaître l'état de leurs voisins,

des communications seront alors nécessaires. Il est indispensable de minimiser le coût des communications pour s'assurer d'une bonne scalabilité.

Les communications sont coûteuses pour plusieurs raisons. Tout d'abord, le temps nécessaire pour effectuer la communication représente presque toujours un coût. Il y a également des cycles CPU qui vont être utilisés pour transmettre les données et non pas pour effectuer des calculs, cela représente également un coût. Les communications nécessitent souvent des synchronisations, il en résulte que des tâches vont se retrouver inactives car elles sont en attentes vis-à-vis des autres tâches, ce coût peut être très lourd s'il est mal géré. Enfin le dernier coût important provient de l'infrastructure réseau en elle-même, le trafic peut se retrouver saturé et la bande passante fortement diminuée.

A l'instar des accès mémoires, le temps d'une communication peut se quantifier en temps de latence et en bande passante[9]. Envoyer de nombreux petits messages peut engendrer une domination des temps de latence sur la bande passante. Il est souvent plus efficace de regrouper ces paquets de petits messages dans un plus grand message afin d'augmenter la bande passante.

Les communications peuvent être explicites ou implicites. Dans une architecture à mémoire partagée, toutes les tâches ont accès à une mémoire commune. C'est ce qui rend cette parallélisation facile à mettre en œuvre. Les communications entre les tâches sont ici transparentes. Dans le cas d'une architecture à mémoire distribuée, les communications sont explicites. Il est souvent nécessaire de mettre en œuvre une infrastructure réseau. Le programmeur doit donc être vigilent, et en particulier veiller à ce que les temps de communications entre les tâches ne soient évidemment pas supérieurs aux gains apportés par la parallélisation.

4.4 Synchronisation des tâches

La programmation parallèle peut être source de problèmes, comme par exemple le cas où plusieurs tâches modifient la même variable partagée en même temps. Le résultat de cette collision est une indétermination de la valeur de la variable. Elle peut soit prendre la

[9] Rappel : le temps de latence est le temps minimum nécessaire à l'envoi d'un message (de zéro octet) d'une tâche à une autre. La bande passante est la quantité de données qui peut être transférée par unité de temps.

valeur donnée par la dernière tâche qui y a eu accès, soit même rester indéterminée. La solution est d'introduire un verrou sur la variable qui empêche son accès simultané par plusieurs tâches. Les tâches concurrentes restent alors en attente de la libération de la variable protégée.

Une autre utilité des synchronisations et de garantir que toutes les tâches soient au même niveau. Typiquement lorsqu'une opération nécessite le déroulement complet de l'opération précédente, une barrière de synchronisation doit être introduite. Toutes les tâches arrivant sur cette barrière se mettent en attente jusqu'à ce qu'elles y soient toutes arrivées. Ainsi la cohérence des données nécessaire à l'opération suivante est préservée. Le programmeur doit rester vigilant à l'utilisation des barrières de synchronisation, en effet un trop grand nombre de tâches en attente nuit à la scalabilité. Ce problème est très sensible sur les architectures massivement parallèles tel le GPGPU.

4.5 Réduction de variables

La réduction est une opération pouvant mener à une baisse des performances, notamment sur les systèmes massivement parallèles. C'est pourquoi il est nécessaire de rester vigilent lors de son utilisation. Soit par exemple une addition vectorielle parallélisée, avec autant de tâches que la dimension des vecteurs à additionner. L'algorithme est très simple, il est illustré à la Figure 17, chaque tâche procède à une seule addition.

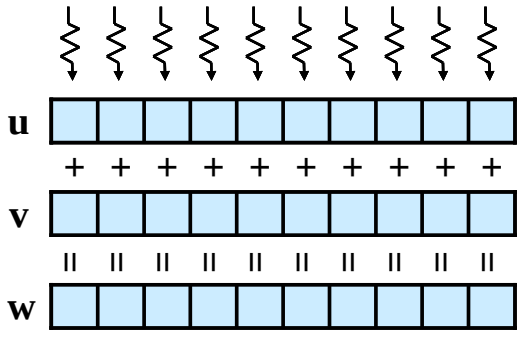


Figure 17 : Addition vectorielle multi tâche.

Soit à présent le produit scalaire entre u et v, toujours en utilisant un maximum de tâches (Figure 18). Les opérations de multiplication sont massivement parallélisables, mais il faut ensuite additionner les résultats de chaque tâche : c'est la réduction. Il est possible de n'utiliser qu'une seule tâche pour cette opération, typiquement une boucle sur n (avec n

le nombre d'éléments dans w) qui additionne le tout. Le nombre d'opérations est alors de n. Il est également possible d'utiliser un schéma de réduction multi tâches. Le nombre d'opérations séquentielles est log(n)/log(2) car à chaque itération le nombre de tâches actives est divisé par deux. Le résultat du produit scalaire est la première entrée du tableau w. Une opération de réduction peut fortement réduire la scalabilité d'une tâche parallèle. En effet, les gains apportés par le parallélisme sur les opérations parallélisées (ici les additions) peuvent être réduits voire perdus lors de la réduction.

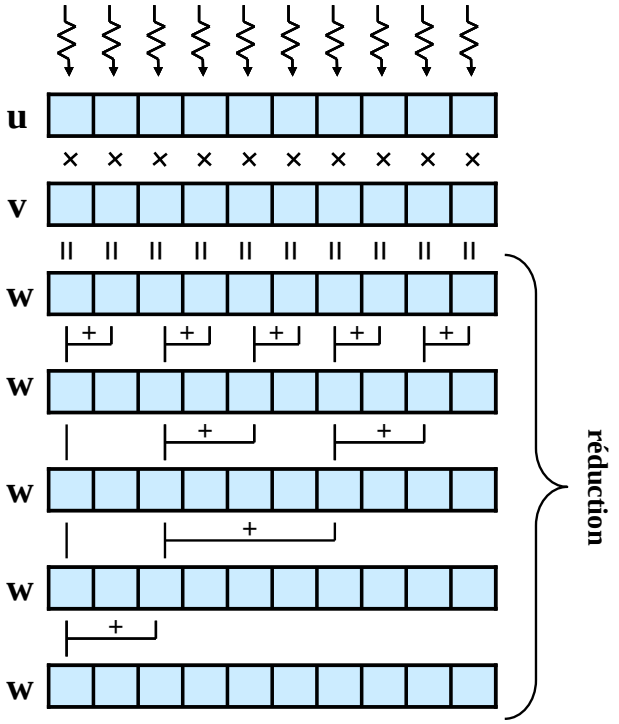


Figure 18 : Produit scalaire multi tâche. Après les multiplications, une opération de réduction est nécessaire.

4.6 Bibliothèques de calcul parallèle

Les quelques bibliothèques présentées ici font partie des plus connues pour réaliser du calcul parallèle.

OpenMP

Pour Open Multi-Processing, c'est une API[10] multi plateforme pour les langages C/C++ et Fortran. Elle permet de réaliser facilement des codes parallèles sur architectures à

[10] Application Programming Interface : interface de programmation.

mémoire partagée. L'opération la plus courante est de paralléliser des boucles à condition que chaque itération dans la boucle soit indépendante des autres.

MPI

Message Passing Interface est également une API pour les langages C/C++ et Fortran. Contrairement à OpenMP, elle exploite les architectures à mémoire distribuée par passage de messages entre les machines. C'est aujourd'hui devenu le standard sur les systèmes parallèles à mémoire distribuée tels que les clusters et les grilles de calcul.

CUDA

Compute Unified Device Architecture est une API développée par Nvidia qui permet d'exploiter les cartes graphiques de la marque pour exécuter des calculs. Cette API sera présentée en détail au Chapitre III.

OpenCL

Open Computing Language est une API développé par le Khronos Group[11] en 2008. D'abord présenté comme une alternative ouverte à CUDA, OpenCL est bien plus car elle gère aussi bien les GPU que le multi-CPU.

5 Calcul scientifique sur processeurs graphiques

5.1 Introduction au GPGPU

5.1.1 Pourquoi le GPGPU ?

Les éditeurs de jeux vidéo souhaitant proposer des jeux toujours plus réalistes, les moteurs physiques de dernières générations demandent une puissance de calcul colossale. Les cartes graphiques, sur lesquelles repose l'exécution de ces jeux, ont dû évoluer et permettent maintenant non seulement de l'affichage d'objets 3D mais également d'effectuer des opérations arithmétiques. De plus, compte tenu de la puissance de calcul demandée, les GPUs sont devenus plus puissants que les CPUs traditionnels (Figure 19). La différence est particulièrement flagrante pour le calcul en simple précision.

[11] Le Khronos Goup est un consortium d'industriels dans le but de concevoir des standards ouverts.

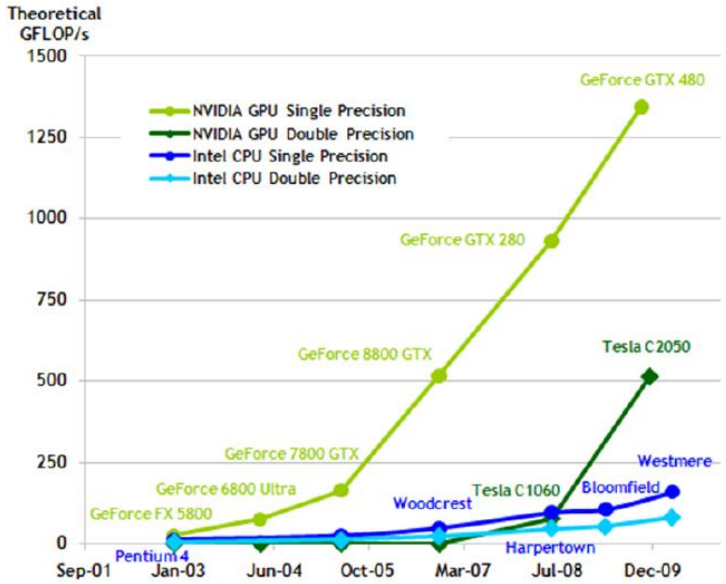


Figure 19 : Puissance des GPUs par rapport aux CPUs traditionnels. Source [Nvidia 11].

La grande puissance des cartes graphiques est exploitée depuis une dizaine d'années par la communauté scientifique. Les premiers calculs sur GPU étaient effectués en détournant les API graphiques. Depuis 2007, les cartes graphiques sont programmables grâce à l'introduction de CUDA par Nvidia. Ce langage dérive du C/C++ et permet de gérer l'architecture particulière des cartes graphiques [Nvidia 11].

Un autre attrait au GPGPU est le coût moindre à l'achat d'une carte graphique par rapport à un serveur de calcul. Par exemple au G2Elab, le serveur de calcul de l'équipe modélisation (DELL PowerEdge R610, 2 processeurs de 4 cœurs chacun) vaut environ 8000€ (achat effectué en 2009) tandis que la carte graphique exploitée durant ma thèse, Nvidia Tesla C1060 vaut environ 1000€ (2010). Une carte graphique d'entrée de gamme, équipée par exemple d'une puce Nvidia Geforce G210 avec 512 Mo de mémoire, coûte environ 50€ (2012). Le rapport performance/coût permit à l'architecture GPGPU de faire l'objet de travaux dans notre communauté des méthodes intégrales basses fréquences [Lezar 10] et dans d'autres également tel par exemple celle des éléments finis [Oliveira Rodrigues 2012].

Les cartes graphiques sont puissantes et peu onéreuses, alors pourquoi ne pas les utiliser pour tous les calculs ? Malgré les avantages cités précédemment, l'architecture particulière des GPUs restreint leur utilisation au calcul massivement parallèle (milliers de

threads). De plus, le calcul sur GPU n'est vraiment intéressant qu'en simple précision (Architecture GT200, 2008). Un dernier point est que les opérations arithmétiques ne sont pas toujours aux mêmes standards que ceux sur CPU. En d'autres termes, il peut y avoir une différence (sur les dernières décimales en général) entre les résultats de deux opérations identiques effectuées l'une sur GPU et l'autre sur CPU.

5.1.2 Présentation de la Nvidia Tesla C1060

Dans l'objectif de réaliser des expériences en GPGPU, le laboratoire a investi dans une carte graphique dédiée au calcul scientifique. Ce calculateur, la Nvidia Tesla C1060 (Figure 20), promet une puissance de calcul brute de l'ordre du téraflop (1000 milliards d'opérations en virgules flottantes par seconde).

Figure 20 : Nvidia Tesla C1060.

La Tesla C1060 est de capacité de calcul 1.3 dans la classification de Nvidia, cela signifie qu'elle permet des calculs en double précision (norme IEEE 754). Cependant la double précision n'est pas très performante, un calcul en double précision est environ 8 fois plus lent que son équivalent en simple précision. Les calculs seront effectués de préférence en simple précision, à condition bien sûr que l'utilisation de cette dernière n'introduit pas plus d'erreur que les mêmes calculs en double précision. La validité des calculs en simple précision des méthodes intégrales sera présentée au chapitre suivant (paragraphe 5.4).

D'un point de vue composant, la Tesla C1060 possède 240 cœurs de calculs, la fréquence d'horloge est de 1,3 GHz. La quantité de mémoire RAM dédiée est de 4Go en DDR3 (102 GB/sec).

5.1.3 GPGPU : le CPU assisté par le GPU

Les cartes graphiques ne sont pas autonomes, elles sont toujours pilotées par un ordinateur qui est appelé l'hôte. L'hôte peut transférer des données entre sa propre mémoire vive et celle du GPU, et il peut également envoyer des programmes à exécuter qui sont appelés des kernels.

Classiquement, un programme GPGPU sera construit de la façon suivante :

1. Allocation de la mémoire sur le GPU
2. Transfert des données de l'hôte vers le GPU
3. Exécution d'un ou plusieurs kernels
4. Transfert du résultat du GPU vers l'hôte
5. Libération de la mémoire allouée sur le GPU

En plus du rôle de chef d'orchestre, l'hôte effectue également toutes les opérations restées séquentielles.

5.2 Architecture d'un GPU

5.2.1 Organisation des cœurs de calcul sur un GPU

La Figure 21 présente le schéma de l'architecture d'un GPU [Nvidia 11]. La différence avec l'architecture PC (Figure 2) est flagrante. Tout d'abord c'est une architecture à mémoire partagée, tous les cœurs de calcul ont accès à une RAM commune. Le GPU possède de très nombreux cœurs de calcul. C'est une architecture massivement parallèle, nécessaire au rendu de l'affichage sur un écran d'ordinateur. C'est pourquoi il y a d'avantage de transistors dédiés au calcul qu'au contrôle ou la mise en cache de données. Les cœurs de calcul sont regroupés dans des blocks appelés multiprocesseurs. Ces multiprocesseurs contiennent un contrôleur et un ou plusieurs niveaux de mémoire cache. Une difficulté qui s'annonce déjà sera de gérer les flux dans les caches car ici, contrairement à une architecture multicœur sur PC, tous les cœurs n'ont pas accès aux mêmes mémoires caches.

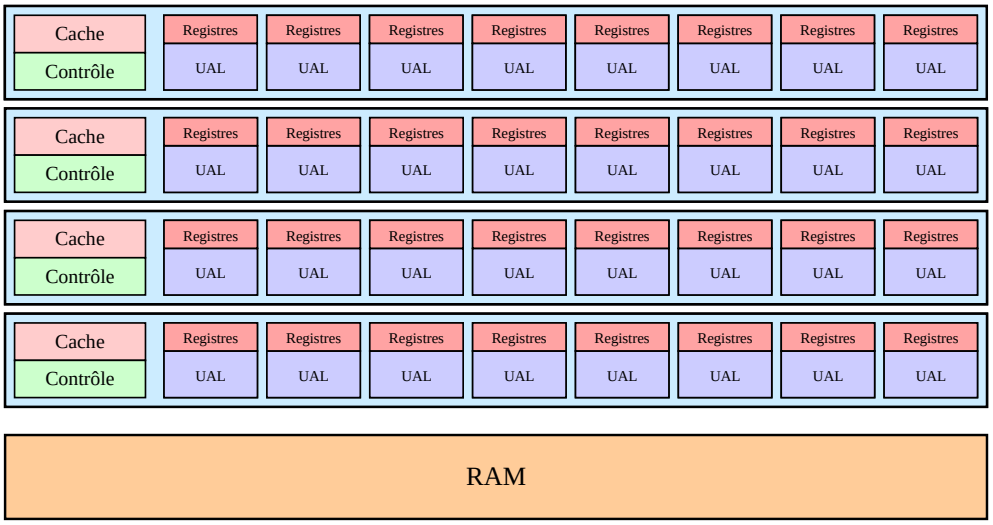


Figure 21 : Architecture d'un GPU.

5.2.2 Niveaux de mémoires sur un GPU

Comme les PCs, les cartes graphiques disposent également de plusieurs niveaux hiérarchisés de mémoire (Figure 22). Il y a une mémoire RAM, appelée ici mémoire globale, de quantité de l'ordre de la centaine de Mo au Go. Il existe également des mémoires de constantes et de textures[12], de quelques dizaines de Ko chacune. Elles peuvent être exploitées par les kernels uniquement en lecture, elles sont utilisées pour stocker des données constantes. Ces mémoires RAM, de contentes et de textures sont accessibles par tous les cœurs de calcul. Mais ce sont également les mémoires les plus lentes [Nvidia 11].

Chaque multiprocesseur dispose de mémoires qui lui sont propres (accessible uniquement par ce dernier). La mémoire partagée, de l'ordre de 128 Ko, est accessible par tous les cœurs d'un multiprocesseur. Ainsi, seuls les threads s'exécutants dans le même multiprocesseur pourront communiquer rapidement. Enfin, chaque cœur de calcul dispose d'une mémoire cache et de registres (16 Ko par cœur). Les communications entre les cœurs et les mémoires caches sont beaucoup plus rapides qu'avec la mémoire globale. Il faudra donc veiller à les utiliser le plus possible. Une autre remarque est que les mémoires caches sont très limitées en quantité, voilà pourquoi les calculs doivent rester suffisamment

[12] En imagerie de synthèse, une texture est un motif de base qui est utilisé pour habiller un objet 2D ou 3D. Les textures sont stockées sur les cartes graphiques dans une mémoire spécifique.

léger. En effet, dans le cas d'un calcul demandant plus de mémoire que celle disponible localement, le programme est obligé d'adresser de la mémoire globale, ce qui ralentit considérablement l'exécution de ce dernier.

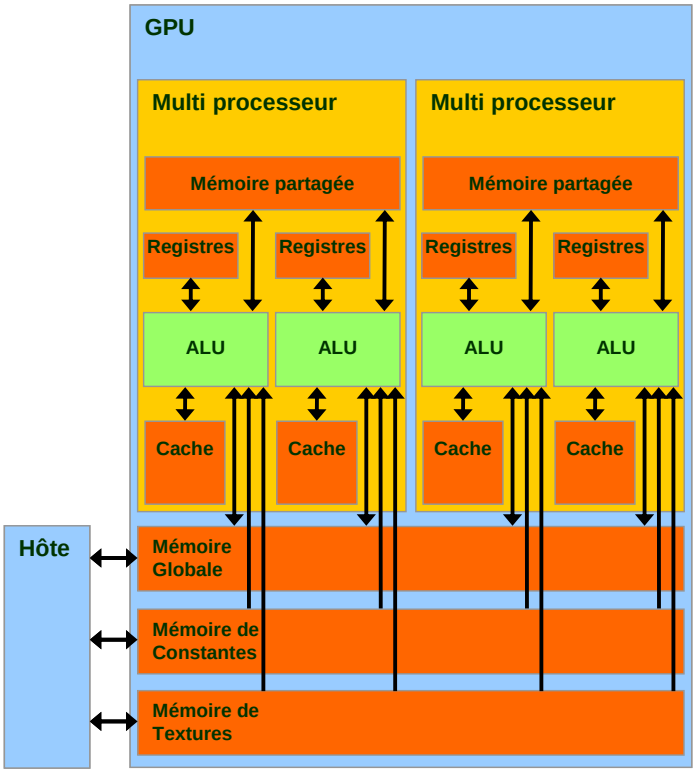


Figure 22 : Hiérarchisation des mémoires dans un GPU. Source de l'illustration : Nvidia.

5.2.3 Architecture SIMT

L'architecture SIMT pour Single Instruction Multiple Thread est une variante multithreadée du SIMD de la classification de Flynn. Cela signifie que tous les threads vont exécuter la même instruction sur des données différentes. De plus, comme pour l'architecture SMT, il y a d'avantage de threads qui sont exécutés simultanément que de cœurs de calcul. Les instructions sont exécutées par paquets de 32 threads sur chaque processeur multicœur. Ces paquets indivisibles de threads sont appelés des warps. La Tesla C1060 dispose de 30 processeurs de 8 cœurs chacun, c'est-à-dire que 30 warps sont exécutés simultanément. Donc pour exploiter pleinement ce calculateur, des multiples de 30x32=960 threads doivent être exécutés ! Il est alors nécessaire d'effectuer des tâches massivement parallélisables.

5.2.4 Coût des opérations arithmétiques

La Figure 23 présente le coût des opérateurs arithmétiques [Nvidia 11] des cartes graphiques Nvidia de capacité 1.3 (Tesla). Les résultats sont pondérés sur la base qu'une addition coûterait 1 cycle d'horloge. Comme pour les architectures CPU, la division est très coûteuse, donc à limiter. Une information essentielle est ici la faible performance du calcul en double précision. En effet, chaque multicoeur ne dispose que d'une seule unité de traitement 64 bits. Le coût de la division en double précision n'est pas fourni dans les documents techniques de Nvidia.

Opération		Simple précision	Double précision
Nombre à virgule flottante	Addition, Soustraction	1	8
	Multiplication	1	8
	Division, Modulo	4	-

Figure 23 : Coût des opérateurs arithmétiques des cartes graphiques Nvidia de capacité 1.x.

5.3 Programmation GPGPU

L'architecture GPU étant particulière, le modèle de programmation l'est tout autant. Il prend évidemment compte de l'arrangement des cœurs de calcul, des différents niveaux de mémoires, ainsi que des communications entre le GPU et l'ordinateur hôte. La construction complète d'un petit programme CUDA est proposée dans l'Annexe A.

5.3.1 Compute Unified Device Architecture

CUDA est le langage développé par Nvidia qui permet d'exploiter facilement (c'est-à-dire sans détourner des API graphiques) les cartes graphiques pour effectuer des calculs génériques. Il est supporté nativement par le C/C++ et le Fortran. Le compilateur est nvcc pour Nvidia C Compiler. Il est basé sur gcc pour les systèmes Unix et Microsoft Visual Studio pour les systèmes Microsoft Windows.

Les programmes GPGPU sont envoyés à la carte graphique sous forme de kernels. Ces kernels sont des fonctions qui ne renvoient rien (void). Il est possible de définir des sous fonctions typées, mais il n'est pas possible de les appeler directement. Le mot clef __global__ devant une fonction indique qu'il s'agit d'un kernel. Le mot clef __device__

indique que la fonction s'exécute sur la carte graphique. Il est également possible d'utiliser les mécanismes d'inlining[13] du C++.

5.3.2 Topologie de la grille de calcul

La topologie d'une grille de calcul CUDA dérive de l'architecture GPU (Figure 24). Les threads sont groupés dans des blocs. Les threads d'un même bloc partagent une mémoire dite mémoire partagée. Ce n'est pas sans rappeler que chaque multiprocesseur contient une mémoire cache à laquelle tous ses cœurs ont accès. Ces blocs contiennent au maximum 512 threads (soit 16 warps). Le nombre de threads par bloc est défini par le développeur. Ce dernier doit veiller à ce que chaque thread ait les ressources mémoires nécessaires. En effet, plus un thread sera gourmand en mémoire et moins il sera possible d'en exécuter sur un seul multiprocesseur. Il est également déconseillé d'avoir moins de 32 threads (1 warp) par bloc, car dans ce cas les multiprocesseurs ne seront pas exploités complètement. Les blocs peuvent être définis en 1D, 2D, et même en 3D en fonction de la complexité du kernel. Chaque thread possède des coordonnées locales uniques au sein du bloc qu'il occupe. Chaque bloc possède également des coordonnées uniques dans la grille de calcul.

La Figure 24 représente une grille de calcul CUDA à une dimension, les blocs sont groupés sur un seul axe, ainsi que les threads dans chaque bloc. Cette topologie est bien adaptée aux opérations vectorielles. En effet, chaque thread effectue une opération sur un seul élément du vecteur. L'indice de l'élément du vecteur de chaque thread est déterminé par la coordonnée locale du thread dans le bloc et par la coordonnée du bloc dans la grille multipliée par la taille du bloc. Cette topologie permet un accès coalescent à la mémoire globale au sein de chaque bloc. En effet, chaque threadIdx +1 accède à l'entrée du tableau consécutive de threadIdx. La Figure 25 illustre une grille de calcul en 2 dimensions adaptée à une opération matricielle. De manière analogue à l'exemple précédent, les coordonnées de la matrice sont associées aux coordonnées des threads et des blocs dans la grille de calcul.

[13] En C++, une sous fonction précédée du mot clef inlining va être recopiée par le compilateur aux endroits où elle est appelée. L'exécutable final sera plus volumineux que la version initiale (avec appels de fonctions) mais également plus rapide car justement il n'y a plus d'appels de fonctions.

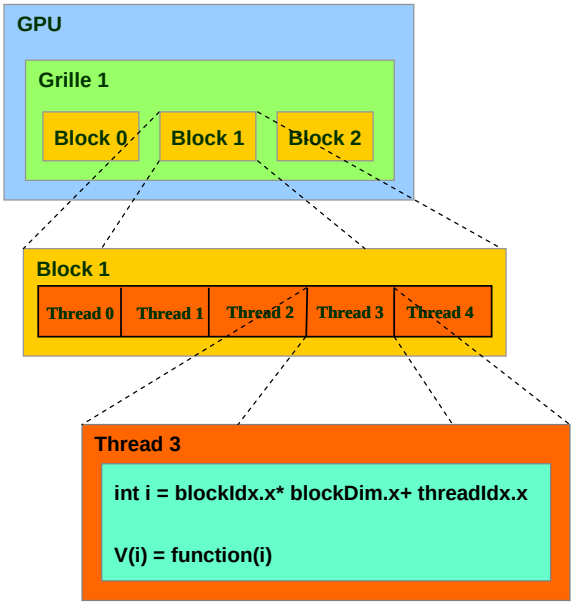


Figure 24 : Grille de calcul CUDA en 1 D.

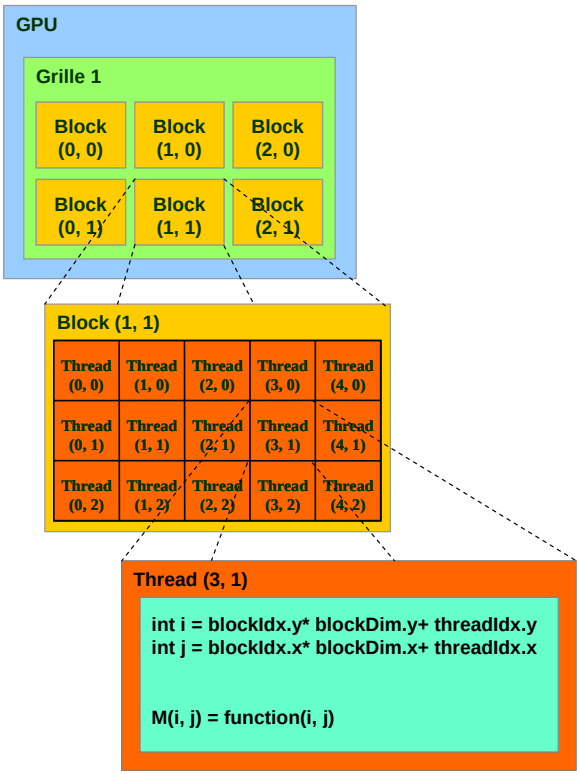


Figure 25 : Grille de calcul CUDA en 2D.

Il est bien sûr possible de définir des grilles à une dimension avec des blocs 2D, toutes les combinaisons sont possibles. Le choix de la topologie dépend du problème à paralléliser. Les deux exemples présentés sont les plus typiques car les opérations vectorielles et matricielles sont les opérations arithmétiques les plus courantes. Une dernière remarque, le langage CUDA tout comme le Fortran utilise le format colonne pour définir les tableaux à plus d'une dimension. Par conséquent les blocks et les threads sont coalescents dans le sens des colonnes. Il est alors nécessaire, lorsque le format ligne pour le stockage des matrices est utilisé, de transposer la grille de calcul.

5.3.3 Transfert des données

La mémoire sur la carte graphique est allouée de manière analogue au C/C++, en utilisant la fonction cudaMalloc. Il n'est pas possible d'allouer la mémoire dynamiquement, c'est-à-dire pendant l'exécution d'un kernel. Il est donc nécessaire de prévoir les ressources mémoires nécessaires avant l'exécution d'un calcul.

Le transfert des données est effectué explicitement entre la RAM de l'hôte et celle du GPU. Comme pour toute forme de communication, il est nécessaire de limiter les temps de latence et de maximiser la bande passante. Les données sont alors transférées de préférence sous forme de tableaux.

5.3.4 Mémoire partagée

Il est parfois nécessaire d'utiliser de la mémoire partagée lorsque plusieurs threads d'un même block ont besoin de partager des données [Nvidia 08]. Soit le problème suivant : l'interversion des éléments d'un tableau deux à deux. L'objectif primordial est de ne pas rompre la coalescence lors des accès à la mémoire globale. Le kernel CUDA correspondant est illustré à la Figure 26.

La grille choisie est une grille à une dimension dans laquelle chaque thread effectuera une opération d'inversion. N/2 threads sont donc définis, avec N la taille du tableau à traiter. La première opération (Figure 26, 1) est l'extraction coalescente des données du tableau. Une barrière de synchronisation est placée, cette dernière est indispensable car il est nécessaire que tous les threads aient fini la lecture depuis la mémoire globale avant de passer à la suite. En d'autres termes, le tableau partagé doit être totalement initialisé avant

de commencer les interversions. Les données sont ensuite interverties (Figure 26, 2), puis nouvelle synchronisation. Finalement, le résultat est retourné dans le tableau d'origine de façon coalescente (Figure 26, 3). Le programme est proposé dans l'Annexe A.

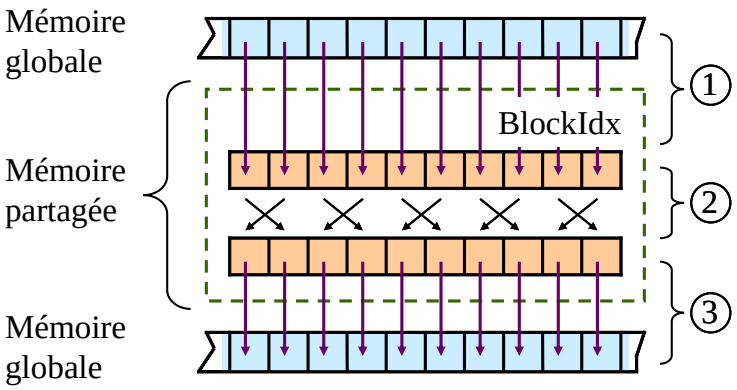


Figure 26 : Exemple de programme CUDA nécessitant l'utilisation de mémoire partagée afin de ne pas rompre la coalescence de l'accès à la mémoire globale.

Cet exemple très simple illustre comment utiliser la mémoire partagée et également l'importance des synchronisations. Une dernière remarque, tout comme la mémoire partagée, les synchronisations ne s'appliquent qu'aux threads d'un même block.

5.3.5 Calculateur d'occupation GPU

Nvidia fournit un outil (un fichier Microsoft Excel) qui permet d'estimer les performances d'un calcul CUDA en fonction de la mémoire utilisée. La première étape demande la version de l'architecture GPU, ici 1.3. La deuxième étape demande la configuration de la grille de calcul et les informations sur l'utilisation des mémoires du GPU fournies par le compilateur. Les estimations des performances du code CUDA sont ensuite affichées.

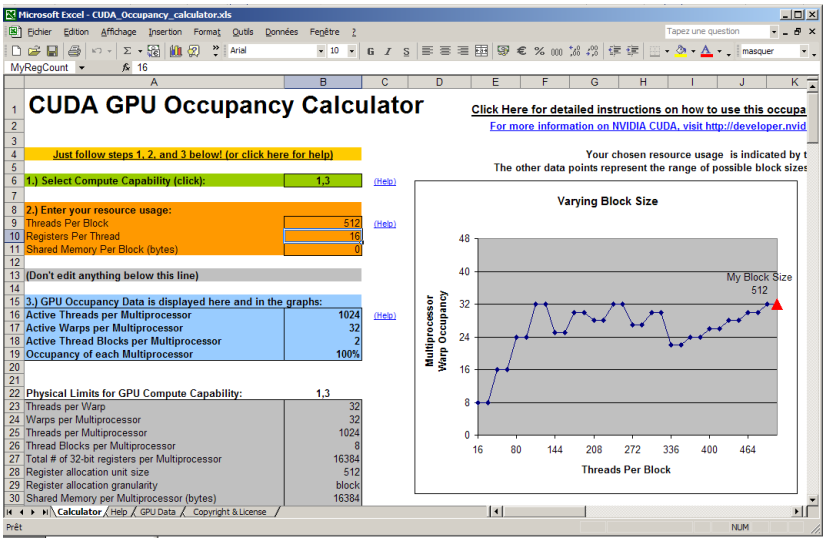


Figure 27 : Calculateur d'occupation GPU de Nvidia.

Ces estimations sont réalisées grâce à des courbes empiriques réalisées par Nvidia. Attention, ce calculateur ne tient pas compte des éventuels transferts avec la mémoire globale, donc un accroissement de l'occupation de chaque multiprocesseur ne signifie pas forcément un accroissement des performances.

5.3.6 JCuda

JCuda est une bibliothèque développée par Marco Hutter [Hutter] qui permet de piloter CUDA depuis des applications Java. Cette API contient la plupart des méthodes disponibles sur CUDA. L'intérêt est d'obtenir des codes GPGPU nativement multiplateformes (à condition que l'utilisateur ait installé les bibliothèques correspondant à son architecture). Les kernels CUDA sont compilés automatiquement lors de leurs appels. CuBlas [Nvidia 12] et les autres bibliothèques de calcul intensif sont également supportés.

6 Evolutions des architectures HPC

Les architectures HPC sont en évolution constante. Cette section propose une veille technologique sur les tendances actuelles et à venir.

6.1 Architectures HPC en 2012

Depuis notre acquisition du calculateur Tesla C1060 en 2010, Nvidia a fait considérablement évoluer ses architectures GPGPU et a ouvert son compilateur CUDA. La

nouvelle architecture GPU du nom de code Fermi, en référence au physicien nucléaire Enrico Fermi, est le successeur de la génération GT200 [Nvidia 09]. Les principales nouveautés sont :

- Un accroissement considérable de la puissance de calcul (x10 d'après Nvidia). Les cœurs sont plus nombreux (512) et la fréquence d'horloge plus élevée. Les multiprocesseurs contiennent 32 cœurs au lieu de 8 ! Les calculs en double précision sont également mieux pris en charge.
- L'apparition de niveaux de cache L1 et L2 génériques comme sur les CPUs classiques au lieu des caches de textures. Nvidia a donné la priorité au calcul plutôt que les jeux vidéo comme le fait classiquement la concurrence. Le niveau de cache L2 permet de lever une partie des problèmes de coalescence !
- La possibilité d'exécuter plusieurs kernel CUDA simultanément.
- Le support des mécanismes de détection et correction des erreurs dans la mémoire (ECC), comme sur la RAM des PCs.
- Un meilleur support de Microsoft Visual Studio, avec des outils de débogage améliorés.

Le coût des opérations arithmétiques, toujours sur la base qu'une addition coûterait 1 cycle d'horloge, est donné à la Figure 28 pour une carte graphique Nvidia de capacité 2.0 [Nvidia 11]. La grande différence avec l'architecture GT200 est la meilleure gestion des opérations en double précision. En effet, le ratio en temps de calcul est ici de 1/2 entre un calcul en simple et le même en double précision.

Opération		Simple précision	Double précision
Nombre à virgule flottante	Addition, Soustraction	1	2
	Multiplication	1	2
	Division, Modulo	8	-

Figure 28 : Coût des opérateurs arithmétiques des GPU Nvidia de capacité 2.0.

Face à la monter en puissance du standard OpenCL, Nvidia a annoncé en décembre 2011 une ouverture de son compilateur CUDA [TomsHardware]. Ce nouveau compilateur est basé sur l'infrastructure LLVM[14] [LLVM] qui est une infrastructure de compilateur conçue pour optimiser la compilation, l'édition de liens, l'exécution et les temps morts dans un programme écrit dans un langage quelconque. CUDA est aujourd'hui limité au C/C++ et au Fortran, mais pourrait désormais s'ouvrir à d'autres langages tels que le Java ou le Python.

6.2 Tendances à venir

L'avenir des architectures CPU peut sembler sombre car il devient difficile de monter en puissance. En effet il est difficile de graver les transistors toujours plus finement dans le silicium. Cependant Intel annonce une évolution appelée Tri-Gate qui permet une gravure des transistors en trois dimensions. Ce procédé permettrait des gains de performances. La gravure sur silicium semble néanmoins condamnée à plus ou moins court terme. La relève pourrait être assurée par des processeurs en graphène, ce semi-conducteur à base de carbone permettrait de graver bien plus finement qu'il est possible actuellement. La commercialisation de ces nouveaux processeurs n'est pas attendue avant 2025.

La solution actuelle à la difficulté de graver plus finement est la multiplication des cœurs de calcul. L'Intel Many Integrated Core (MIC) représente la nouvelle stratégie d'Intel [Intel 11a] dans le marché du HPC. Il s'agirait d'une carte PCI Express embarquant au minimum une cinquantaine de cœurs de calcul. L'avantage de cette architecture est qu'il ne sera pas nécessaire à priori de recompiler ses codes pour les utiliser sur cette machine. Intel propose avec le MIC une solution intermédiaire entre le GPGPU et le multicœur. Le niveau de parallélisation ne sera pas aussi important qu'avec le GPGPU, mais les calculs peuvent être bien plus sophistiqués du fait de l'utilisation des génériques processeurs x86. La sortie est prévue pour fin 2012.

Nvidia de son côté continue de faire évoluer ses architectures HPC à base de processeurs graphiques. L'année 2012 devrait voir sortir les premiers calculateurs construits sur la récente architecture Kepler.

[14] Low Level Virtual Machine

7 Conclusion du chapitre

La fin des années 90 et le début des années 2000 ont été marqués par la course à la puissance de la part des constructeurs de processeurs. Cependant il est maintenant difficile de graver les transistors plus finement. La solution pour que les ordinateurs continuent de monter en puissance est d'utiliser plusieurs processeurs : c'est le parallélisme. Les algorithmes existants vont devoir être adaptés à l'utilisation de plusieurs processeurs.

Ce chapitre a présenté les différents types de parallélisme, et les différentes architectures parallèles. La programmation dépend fortement de l'architecture parallèle choisie. Les enjeux du calcul parallèle sont le partitionnement du problème sur les différentes tâches, la gestion des ressources, les communications entre les tâches, les synchronisations, et la réduction de variables. Il est nécessaire de comprendre le comportement des programmes afin de les adapter au mieux à une architecture parallèle.

Les processeurs graphiques, de par le nombre de cœurs de calcul qu'ils embarquent, sont devenus des calculateurs très puissants en restant peu onéreux. Leur attrait a fortement augmenté dès le moment où ils sont devenus programmables grâce à CUDA et à OpenCL. Les particularités de cette architecture, notamment sur la gestion de la mémoire, nécessitent une refonte en profondeur des algorithmes qui y sont adaptés.

La suite de ce mémoire présente le portage des méthodes intégrales sur plusieurs architectures parallèles. Tout d'abord sur processeurs multicœurs car ils sont présents dans tous les PCs à l'heure actuelle. Ce parallélisme est simple à mettre en œuvre, en effet il s'agit d'une architecture à mémoire partagée sans communications et synchronisations explicites. Une expérience est également réalisée sur un petit cluster de PCs connectés en réseau. La gestion de la mémoire, des communications et des synchronisations y est plus sensible. Enfin, le cœur de ces travaux de thèse porte sur l'adaptation des méthodes intégrales sur architecture GPGPU. Ici les algorithmes sont entièrement repensés pour tirer profit de cette architecture massivement parallèle.

8 Références

[Amdahl 67] G. Amdahl, « Validity of the single processor approach to achieving large scale computing capabilities », AFIPS spring joint computer conference, 1967.

[Bersini 08] H. Bersini, M-P. Spinette, et R. Spinette, « Les fondements de l'informatique: du bit à l'Internet », 2nd édition, Vuibert, 2008.

[BLAS] Documentation BLAS, http://www.netlib.org, 2012.

[BOINC] Projet BOINC : http://boinc.berkeley.edu/, 2012.

[CIMENT] Projet CIMENT : https://ciment.ujf-grenoble.fr/, 2012

[CPU World] I7 specifications, http://www.cpu-world.com/, 2012.

[Dowd 98] C. Severance, K. Dowd, « High Performance Computing », 2nd édition, O'Reilly Media, 1998.

[Duncan 90] R. Duncan, « A Survev of Parallel Computer Architectures », Control Data Corporation, fev. 1990.

[Flynn 72] M. Flynn, « Some Computer Organizations and Their Effectiveness », IEEE Trans. on computers vol. c-21, no. 9, sep 1972.

[Flynn 95] M. Flynn, « Computer architecture : pipelined and parallel processor design », Hardcover, 1995.

[Golub 83] G. Golub, C. Van Loan, « Matrix-computations », North Oxford Academic, 1983.

[Gustafson 88] J. Gustafson, « Reevaluating Amdahl's law », Communications of the ACM, May 1988.

[Hutter] M. Hutter, JCuda : http://www.jcuda.de/

[Intel 11a] Intel, « Intel révèle les détails de la prochaine génération de plateformes haute-performance », communiqué de presse, nov. 2011.

[Intel 11b] Intel, « Intel® 64 and IA-32 Architectures Optimization Reference Manual », juin 2011.

[Kish 02] L. Kish, « End of Moore's law: thermal (noise) death of integration in micro and nano electronics », Physics Letters A, 2002.

[Kumar 94] A. Grama, G. Karypis, V. Kumar, et A. Gupta, « Introduction to Parallel Computing », 1ère édition, Hardcover, 1994.

[Lezar 10] E. Lezar and D.B. Davidson, « GPU Acceleration of Method of Moments Matrix Assembly using Rao-Wilton-Glisson Basis Functions », International Conference on Electronics and Information Engineering, 2010.

[LLVM] Compilateur LLVM : http://llvm.org/

[Lo 97] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm and D. Tullsen, « Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading », ACM Transactions on Computer Systems, vol. 15, vo. 3, August 1997.

[Neumann 47] A. Goldstine, H. Herman, J. Von Neumannn, « Preliminary discussion of the logical design of an electronic computer instrument », 1946.

[Ni 90] X-H. Sun, L. Ni, « Another view on parallel speedup », Supercomputing '90 Proceedings of the 1990 ACM/IEEE conference on Supercomputing, 1990.

[Patterson 03] J. Hennessy, D. Patterson, « Architecture des ordinateurs : une approche quantitative », 3ème édition, Vuibert, 2003.

[Nvidia 12] Nvidia, « Nvidia CUBLAS Library », version 4.1, janvier 2012.

[Nvidia 11] Nvidia, « NVIDIA CUDA C Programming Guide », version 4, juin 2011.

[Nvidia 09] Nvidia, « NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™ », whitepaper, 2009.

[Nvidia 08] Nvidia, « Tutorial CUDA », avril 2008.

[Oliveira Rodrigues 2012] A. W. de Oliveira Rodrigues, « Une méthodologie pour le développement d'applications hautes performances sur des architectures GPGPU : Application à la simulation des machines électriques », Thèse de Doctorat, Université des Sciences et Technologies de Lille, France, 2012.

[Schaller 97] R. Schaller, « Moore's law: past, present and future », Spectrum, IEEE, 1997.

[TomsHardware] Ouverture du compilateur CUDA, http://www.tomshardware.com, 2011.

[Tullsen 95] D. Tullsen, S. Eggers, H. Levy, « Simultaneous multithreading: Maximizing on-chip parallelism », Proceedings, 22nd Annual International Symposium on Computer Architecture, pp. 392-403, 22-24 June 1995.

Chapitre II

Vectorisation d'une formulation intégrale en potentiel pour l'électrostatique

Sommaire

Résumé

Nous développons dans ce chapitre une formulation intégrale en potentiel électrostatique performante et massivement parallélisable dans le but d'être portée sur architecture GPGPU. Cette formulation est tout d'abord vectorisée afin d'accélérer les calculs par l'exploitation du pipelining des processeurs. L'intégration numérique par points de Gauss peut parfois se montrer imprécise voire singulière, nous utilisons alors des solutions analytiques. L'intégration par points de Gauss et l'intégration analytique sont combinées afin d'obtenir le meilleur ratio entre précision et temps de calcul.

1 Introduction

Les méthodes intégrales sont des méthodes de modélisation numériques adaptées aux problèmes d'interactions à distance. Ces méthodes sont donc particulièrement prisées pour modéliser les systèmes électromagnétiques. Nous ne présentons pas toutes les formulations intégrales existantes pour l'électromagnétisme, elles sont nombreuses et très variées. Nous nous intéressons seulement à une formulation en potentiel très simple que nous appliquerons à l'électrostatique. Cette formulation nous sert d'exemple tout au long de ces travaux. Elle est développée dans le but d'être massivement parallélisable, l'objectif final étant le portage sur une architecture GPGPU. Notons que les techniques développées dans ces travaux sont générales et peuvent être pour la plupart applicables à d'autres formulations électromagnétiques telles que la magnétostatique ou la magnétodynamique. L'application à d'autres physiques ou les méthodes intégrales ont fait leurs preuves est évidemment aussi envisageable (gravitation, mécanique, acoustique, électromagnétisme en hyper-fréquences, …).

Ce chapitre débute par une introduction aux méthodes intégrales, nous présentons les caractéristiques principales de ces méthodes et des exemples d'applications en génie électrique. Nous établissons ensuite l'équation intégrale en potentiel pour l'électrostatique tirée de l'identité de Green à partir des équations de Maxwell. Nous comparons plusieurs formulations et sélectionnons la plus performante en vue d'être vectorisée et parallélisée. Nous terminons par une étude comparative de différentes approches d'intégration numériques par points de Gauss et analytique, ainsi que l'hybridation des deux méthodes.

2 Généralités sur les méthodes intégrales

2.1 Introduction aux méthodes intégrales

Les méthodes intégrales font l'objet de nombreux travaux en électromagnétisme, en particulier pour les applications hautes fréquences. Dans les domaines basses fréquences, un groupe s'est constitué au laboratoire autour de ces méthodes, c'est le groupe MIPSE (Modélisation des Interconnexions de Puissance des Systèmes Electriques). Ce groupe est

constitué d'une dizaine de membres en collaboration avec la société CEDRAT qui commercialise le logiciel InCa3D. Nous présentons rapidement ce logiciel dans les sous-parties suivantes ainsi que d'autres exemples de travaux réalisés par méthodes intégrales dans le groupe MIPSE. Ces travaux sont développés en langage Java et mutualisés sur une plateforme commune. Nous discuterons également dans cette partie de la pertinence du développement de nos projets en langage Java.

2.1.1 Logiciel InCa3D

Le logiciel InCa3D [CEDRAT] modélise les connexions électriques des dispositifs du génie électrique avec pour enjeu la compatibilité électromagnétique (CEM). Il repose sur la méthode PEEC (Partial Element Equivalent Circuit) qui permet la synthèse de circuits équivalents [Ruehli 74]. Ces circuits équivalents sont ensuite intégrés dans des solveurs circuits afin de permettre des modélisations « système » globales. Ce logiciel est particulièrement pertinent pour la modélisation de différents dispositifs en particuliers les convertisseurs statiques d'électronique de puissance (Figure 1). Les éléments de circuits équivalents sont les résistances, les inductances et les capacités. Des méthodes intégrales sont utilisées pour les calculer [Ardon 10a]. C'est dans le contexte du logiciel InCa3D que sont réalisés ces travaux.

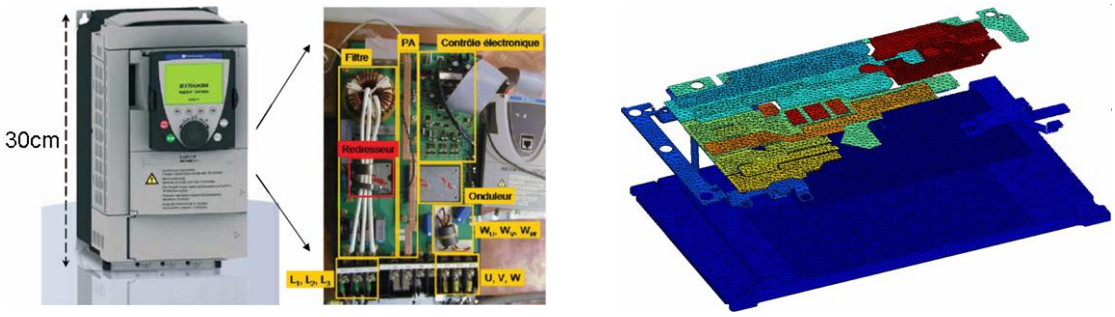


Figure 1 : Variateur de vitesse ATV71 commercialisé par STIE et le maillage à droite des conducteurs en vue de l'extraction des capacités parasites.

2.1.2 Autres exemples de formulations

Les méthodes intégrales peuvent aussi résoudre des problèmes en magnétostatique comme par exemple le calcul de l'aimantation de la coque d'un navire ou d'un sous-marin sous l'effet du champ magnétique terrestre (Figure 2) [Chadebec 01]. Elles peuvent également permettre le calcul de courants induits dans les blindages ou dans les structures

volumiques. Ces formulations intégrales en magnétodynamique sont présentées dans les travaux de thèse de Tung Le Duc [Le Duc 11].

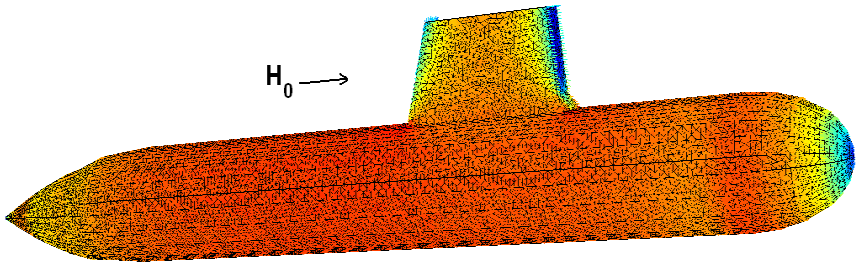


Figure 2 : Aimantation de la coque d'un sous-marin sous l'influence du champ magnétique terrestre.

Un dernier exemple, les méthodes intégrales ont également été utilisées pour modéliser des champs électromagnétiques induits par les courants de corrosions à la surface des coques des navires [Guibert 09].

2.1.3 Pourquoi du calcul scientifique avec Java ?

Le choix du langage Java peut sembler surprenant dans un contexte de calcul hautes performances. Les codes de calcul scientifique sont généralement développés en C/C++ (voire en Fortran). Ce langage bénéficie de bibliothèques de calcul performantes telles que BLAS, LAPACK, etc. Le Java offre cependant quelques avantages intéressants : c'est un langage robuste, qui bénéficie d'une gestion des exceptions de qualité d'où une bonne efficacité et rapidité de développement. Le Java est également multiplateforme, il n'est pas nécessaire de recompiler les codes pour chaque architecture. De plus, les performances de calcul sont tout à fait correctes [Reinauer 11].

Nous proposons de comparer les performances des langages C++ (compilation sur gcc) et Java (JVM d'Oracle) sur un calcul de multiplication matricielle. Nous implémentons l'algorithme classique et sa version vectorisée (voir Chapitre I), tous deux programmés à l'identique sous les deux langages. Nous comparons les temps de calcul pour deux tailles de matrices (Figure 3). Nous observons que les temps de calcul sont quasiment identiques avec l'algorithme classique. Par contre, l'algorithme vectorisé semble être plus rapide en Java qu'en C++. Le compilateur gcc est ici configuré sans options particulières à l'architecture matérielle ce qui peut expliquer cette si grande différence dans les temps de

calcul. Néanmoins ce test démontre que la machine virtuelle Java est capable de vectoriser des calculs efficacement.

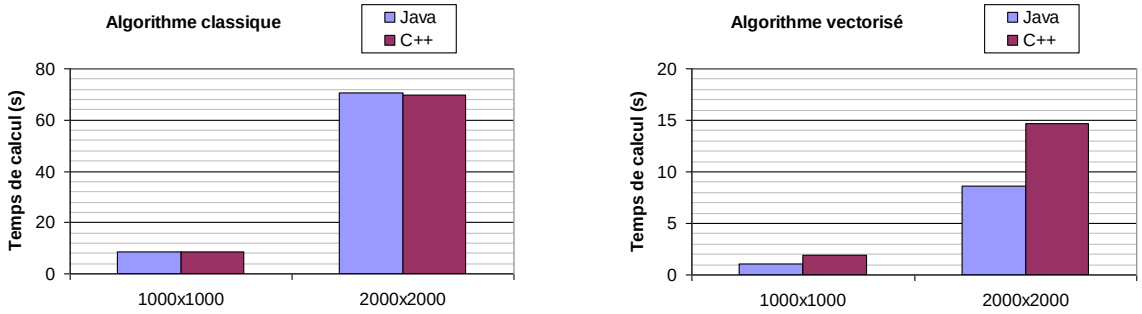


Figure 3 : Comparaison des performances de Java et C++ sur une multiplication matricielle, à gauche l'algorithme classique et à droite sa version vectorisée.

2.1.4 Coût de la programmation objet

La programmation orientée objet possède des avantages indéniables en terme d'évolutivité grâce à la généricité des codes qu'elle procure. Dans le contexte HPC de ce mémoire, il est intéressant d'évaluer le coût en temps de calcul de cette généricité.

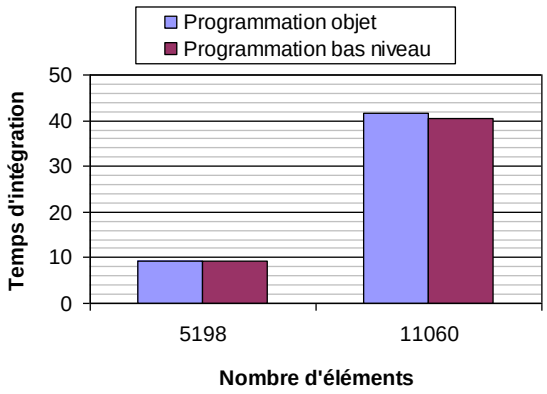


Figure 4 : Temps d'intégration en fonction du nombre d'éléments. Comparaison entre une programmation sur tableaux et une programmation par bibliothèques matricielles.

Le cas test et la formulation intégrale sont ceux présentés dans la section 4.1. Nous comparons à la Figure 4 les temps de calcul entre une programmation fonctionnelle (opérations matricielles effectuées sur des tableaux) et une programmation objet via une bibliothèque de calcul matricielle (opérations matricielles effectuées par des méthodes de la classe matricielle). Nous constatons que la programmation objet est très sensiblement plus lente (<3%) que celle bas niveau. Cependant compte tenu des avantages en terme de

programmation, le coût supplémentaire engendré par la généricité des codes est négligeable.

2.2 Caractéristiques des méthodes intégrales

Les méthodes intégrales et les méthodes par éléments finis sont les deux grandes familles de méthodes fines les plus utilisées. La simulation numérique dans les domaines basses fréquences repose essentiellement sur les méthodes par éléments finis car elles sont très génériques et robustes. Cependant elles ne sont pas toujours les méthodes les mieux adaptées à la modélisation des systèmes électromagnétiques.

2.2.1 Avantages par rapport aux éléments finis

Les méthodes par éléments finis nécessitent le maillage des matériaux inactifs tel que l'air ou le vide. Le maillage de l'air peut être très lourd, comme par exemple la modélisation de circuits imprimés composés de superpositions de conducteurs minces (Figure 5). Par ailleurs, les méthodes par éléments finis posent également le problème de la gestion des infinis ce qui oblige à mailler un volume très important autour du système à modéliser (Figure 6). Une modélisation par éléments finis d'un système électromagnétique génère couramment un nombre de degrés de liberté très élevé dont la plupart proviennent du maillage de l'air.

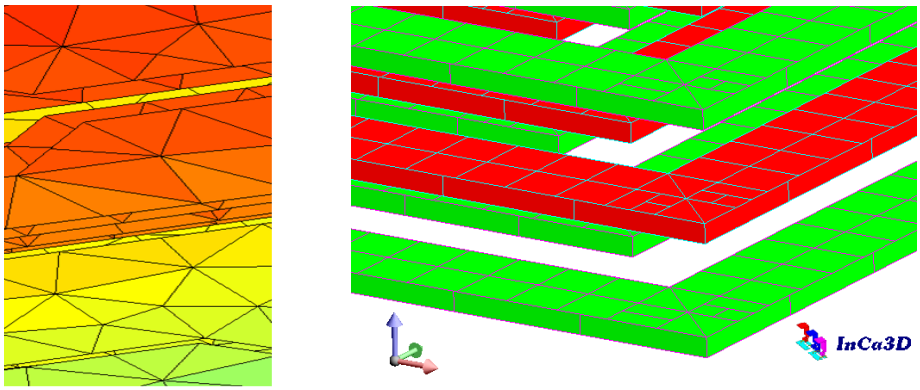


Figure 5 : A gauche, couches superposées de conducteurs minces. A droite, maillage non-conforme de conducteurs.

Les méthodes intégrales sont quant à elles mieux adaptées aux problèmes d'interactions à distance (rayonnement). Les formulations intégrales prennent nativement en compte les infinis et ne nécessitent pas le maillage des matériaux inactifs (l'air dans notre cas). La

réduction du nombre d'inconnues par rapport à une modélisation par éléments finis due à l'absence du maillage des matériaux inactifs est très importante (Figure 6). De plus, l'utilisation des méthodes intégrales facilite grandement les résolutions paramétriques car il n'y a pas d'air à remailler entre chaque simulation. Par ailleurs, dans le cas de matériaux linéaires, certaines formulations intégrales s'affranchissent du maillage de l'intérieur des matériaux, ce sont les intégrales de frontières. Les exemples les plus connus sont basés sur des formulations en potentiel (équation de Laplace). La dimensionnalité du problème est alors réduite car seuls les bords des matériaux sont maillés, ce qui est le point fort des méthodes intégrales de frontières par rapport aux éléments finis. Un dernier point, il est plus facile de se placer à l'ordre zéro avec les méthodes intégrales, et donc d'utiliser des maillages non-conformes (Figure 5), qu'avec les éléments finis.

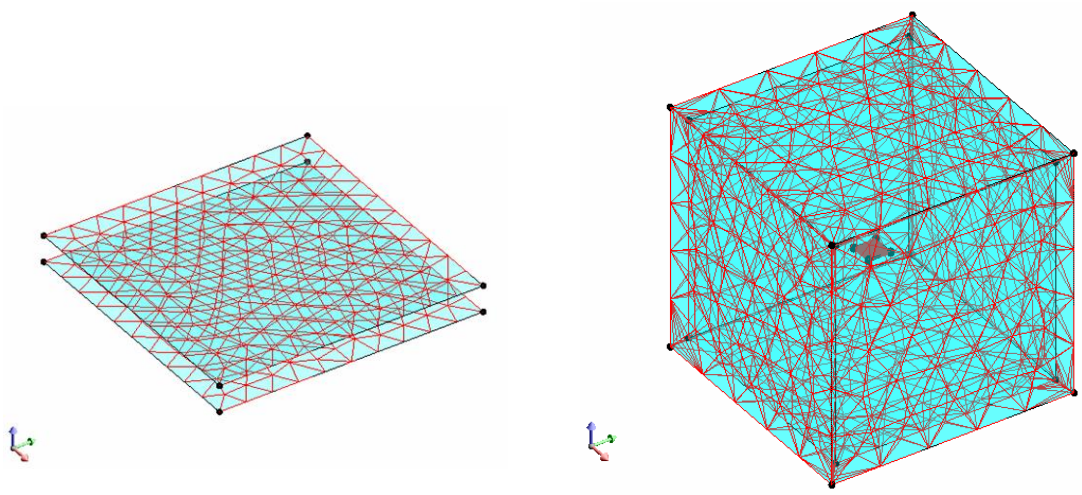


Figure 6 : Modélisation d'un condensateur plan, à gauche par méthodes intégrales (500 éléments surfaciques), à droite par éléments finis (10000 éléments volumiques).

2.2.2 Limitations aux méthodes intégrales

Les méthodes intégrales semblent très séduisantes par rapport aux méthodes par éléments finis pour modéliser les systèmes électromagnétiques. Elles souffrent cependant d'une lacune de taille : ce sont des méthodes de complexité $O(N^2)$ pour l'intégration du système matriciel, avec N le nombre de degrés de liberté, tandis que les méthodes par éléments finis sont de complexité $O(N \log N)$. En effet, les méthodes intégrales sont des méthodes à interactions totales, chaque élément du maillage interagit avec tous les autres et sur lui-même. Par conséquent les méthodes intégrales génèrent généralement des matrices de systèmes d'équations pleines (éléments tous non nuls) qui sont d'une part coûteuses à

calculer en temps processeur et d'autre part difficile à stocker dans la mémoire vive de l'ordinateur. Par exemple, un PC standard en 2012 est équipé d'une quantité de mémoire vive de l'ordre de 8 Go, ce qui limite le nombre de degrés de liberté d'un problème à 30.000 (matrice pleine en double précision). Les méthodes intégrales sont donc incapables en l'état de résoudre un problème de complexité industrielle qui comporte généralement un nombre de degrés de liberté de l'ordre de 100.000, ce qui nécessiterait alors une centaine de Go de mémoire vive !

Par ailleurs, les intégrales sont souvent plus difficiles à calculer que celles rencontrée en éléments finis, que ce soit numériquement ou analytiquement. Pour toutes ces raisons, la communauté basses fréquences s'est éloignée des méthodes intégrales dans les années 80-90 au profit des méthodes par éléments finis malgré le maillage l'air.

2.2.3 Compression des méthodes intégrales

Des techniques de compression ont été développées afin de réduire la complexité des méthodes intégrales. Celles-ci connaissent depuis un renouveau dans notre communauté. Le principe de base de la plupart des méthodes de compression consiste à dissocier les interactions proches des interactions lointaines. Les interactions lointaines peuvent être approximées sans introduire d'erreurs significatives [Greengard 87].

Les méthodes de compression les plus utilisées actuellement sont les FMM pour Fast Multipôle Methods [Greengard 99]. Les interactions lointaines sont approximées par des développements multipolaires basés sur des décompositions en harmoniques sphériques. Le calcul d'un produit matrice vecteur est ici de complexité *O(N)* ce qui a valu à cet algorithme d'être élu comme l'un des dix meilleurs algorithmes du XX^{ème} siècle [Cipra 00]. La complexité globale des FMM en prenant en compte la résolution itérative est de complexité *O(N log N)* comme pour les éléments finis mais avec l'avantage de ne pas considérer les matériaux inactifs. C'est la méthode développée dans le logiciel InCa3D par Vincent Ardon pour le calcul des capacités parasites [Ardon 10b].

D'autres méthodes de compression ont depuis fait leur apparition. La méthode ACA pour Adaptative Cross Approximation [Rjasanow 07] compresse les blocs d'interactions lointaines en les représentant par un produit de deux matrices de rangs inférieurs. Les interactions lointaines peuvent aussi être compressées via une décomposition en ondelettes

[Scheiblich 09], c'est une méthode similaire à la compression d'image JPEG. La compression matricielle par ondelettes sera l'objet du Chapitre IV. C'est une méthode peu invasive et de plus une parallélisation massive performante sur GPGPU est réalisable.

3 Formulation intégrale en potentiel pour l'électrostatique

Considérons un ensemble de conducteurs[1] parfaits (de résistance ohmique nulle) dans le vide (absence de matériaux diélectriques[2]). Un potentiel électrique statique est imposé sur chaque conducteur. Notre objectif est de calculer la distribution de charges se développant sur chaque conducteur et ainsi pouvoir calculer le potentiel et le champ dans tout l'espace. Cette distribution de charge nous donnera également accès aux capacités équivalentes entre chaque conducteur. Ces capacités pourront ainsi être utilisées pour une modélisation par circuit équivalent telle que la méthode PEEC. Notons que si le conducteur est seul dans l'espace ; nous calculerons alors une capacité équivalente entre le conducteur et l'infini.

3.1 Formulation intégrale

3.1.1 Equation de Laplace

Tous les systèmes électromagnétiques sont régis par les équations de Maxwell. Les unités utilisées sont celles du système international (SI). Considérons l'équation de Maxwell-Gauss :

$$\vec{\nabla}.\vec{D} = \rho \tag{1}$$

[1] Un conducteur est un milieu dans lequel des charges mobiles sont susceptibles de se déplacer sous l'effet d'un champ électrique, dans un milieu métallique ces charges sont des électrons.

[2] Un diélectrique est un milieu qui ne contient pas de charges mobiles, mais les atomes ou les molécules qui le constituent sont susceptibles de se polariser sous l'effet d'un champ électrique.

Où $\vec{D}$ est l'induction électrique [C/m²] (ou densité de flux électrique) et ρ est la densité volumique de charges mobiles dans le conducteur [C/m^3]. En l'absence de matériaux diélectriques, l'équation constitutive de la matière est donnée par :

$$\vec{D} = \varepsilon_0 \vec{E} \tag{2}$$

Où ε_0 est la permittivité diélectrique du vide [F/m] et $\vec{E}$ est le champ électrique [V/m]. Ecrivons l'équation de Maxwell-Faraday :

$$\vec{\nabla} \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} \tag{3}$$

Où B est l'induction magnétique [T] et t le temps [s]. Cette équation signifie qu'une variation temporelle du champ magnétique crée un champ électrique. Plaçons nous en régime stationnaire, la dérivée temporelle de (3) s'annule :

$$\vec{\nabla} \times \vec{E} = 0 \tag{4}$$

L'équation (4) permet de définir un potentiel scalaire électrique, tel que :

$$\vec{E} = -\vec{\nabla} V \tag{5}$$

Où V est le potentiel électrique [V]. Un potentiel est défini à une constante près, nous choisissons comme référence un potentiel nul à l'infini. Les équations (1), (2) et (5) permettent alors d'écrire l'équation de Poisson :

$$\vec{\nabla} \cdot (-\varepsilon_0 \vec{\nabla} V) = \rho \tag{6}$$

Plaçons le conducteur à l'équilibre électrostatique, c'est-à-dire qu'après application d'un champ électrique extérieur, les charges se déplacent sous l'effet de ce champ puis occupent une position stationnaire. Cette distribution de charges compense l'effet du champ extérieur, le champ électrique dans le conducteur est alors nul. Par conséquent la densité volumique de charges est également nulle [Durand 66] :

$$\varepsilon_0 \vec{\nabla} . \vec{E} = \rho = 0 \tag{7}$$

Les charges se situent alors sur la surface du conducteur. Nous pouvons écrire l'équation de Laplace :

$$\Delta V = 0 \tag{8}$$

3.1.2 Equation intégrale tirée de l'identité de Green

Considérons le conducteur de volume Ω borné par une surface S présenté sur la Figure 7. P est le point où le potentiel est calculé, Q est un point sur la surface du conducteur, $\vec{r}$ est la distance entre P et Q dirigée vers Q, et $\vec{n}$ est le vecteur normal sortant à la surface du conducteur au point Q.

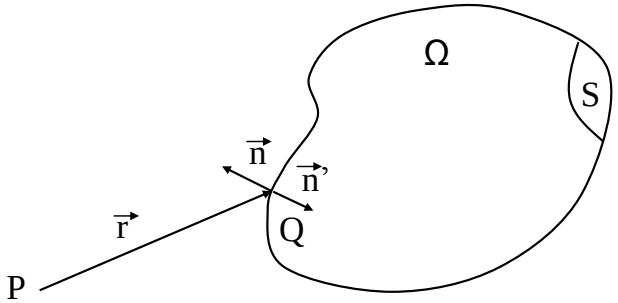


Figure 7 : Conducteur parfait à l'équilibre électrostatique.

Commençons par exprimer le potentiel électrique V dans le volume Ω en fonction des valeurs de V et de sa dérivée normale sur la frontière S du volume Ω. Pour cela écrivons la seconde identité de Green pour un scalaire [Krahenbuhl 83, Haghi-Ashtiani 98] :

$$\iiint_{\Omega} \{G\Delta V - V\Delta G\}\, d\Omega = \iint_{S} \left\{ G\frac{\partial V}{\partial n} - \frac{\partial G}{\partial n} V \right\} dS \tag{9}$$

Où G est une fonction scalaire. Nous choisissons pour G le noyau de Green donné par :

$$G = \frac{1}{r} \tag{10}$$

Il a les propriétés suivantes :

$$\Delta G = 0 \quad \forall r \neq 0, \quad \iiint_{\Omega} \Delta G\, d\Omega = \begin{cases} 1 & \text{si } \Omega \text{ contient r} = 0 \\ 0 & \text{sinon} \end{cases} \tag{11}$$

Le laplacien de G est alors une distribution de Dirac volumique. Nous obtenons :

$$h_0 V(P) = -\iiint_\Omega G\,\Delta V\, d\Omega + \iint_S \left\{ G\frac{\partial V}{\partial n} - \frac{\partial G}{\partial n} V \right\} dS \tag{12}$$

Avec $h_0 = 4\pi \begin{Bmatrix} 1 \\ 1/2 \\ 0 \end{Bmatrix}$ si le point P est situé respectivement à l'intérieur du conducteur, sur sa surface ou à l'extérieur [Durand 64]. Nous pouvons écrire l'équation intégrale associée au laplacien :

$$h_0 V(P) = \iint_S \left\{ G\frac{\partial V}{\partial n} - \frac{\partial G}{\partial n} V \right\} dS \tag{13}$$

Nous remplaçons G par son expression [Durand 64] :

$$h_0 V(P) = \iint_S \left\{ \frac{1}{r}\frac{\partial V}{\partial n} - \frac{\vec{r}.\vec{n}}{r^3} V \right\} dS \tag{14}$$

Nous passons au problème extérieur, c'est-à-dire que nous considérons l'ensemble de l'univers dans lequel nous extrayons le conducteur. La normale sortante devient entrante ($\vec{n}' = -\vec{n}$), d'où le changement de signe à l'équation (14) qui devient [Durand 64] :

$$h_0 V(P) = -\iint_S \left\{ \frac{1}{r}\frac{\partial V}{\partial n} - \frac{\vec{r}.\vec{n}}{r^3} V \right\} dS \tag{15}$$

Plaçons nous sur la surface du conducteur, la relation intégrale devient :

$$2\pi V(P) = -\iint_S \left\{ \frac{1}{r}\frac{\partial V}{\partial n} - \frac{\vec{r}.\vec{n}}{r^3} V \right\} dS \tag{16}$$

V est imposé constant sur la surface du conducteur, le terme V du second membre peut donc sortir de l'intégrale et il ne reste alors que l'intégrale de l'angle solide sous lequel le point P voit le volume chargé (demi hémisphère [Durant 64]) :

$$\iint_S \frac{\vec{r}.\vec{n}}{r^3} = -2\pi \tag{17}$$

Nous obtenons finalement l'équation intégrale suivante qui lie le potentiel à la surface du conducteur à sa dérivée normale [Durand 66] :

$$V(P) = -\frac{1}{4\pi} \iint_S \frac{1}{r} \frac{\partial V}{\partial n} dS \tag{18}$$

3.1.3 Densité surfacique de charges

Les lignes de forces sont normales aux surfaces équipotentielles, donc le champ est normal au voisinage immédiat d'une surface chargée, nous pouvons écrire [Durand 66] :

$$\vec{E}_{(+)} = E_{(+)}.\vec{n} \tag{19}$$

Où $\vec{E}_{(+)}$ est le champ électrique du coté extérieur. De la même manière nous définissons $\vec{E}_{(-)}$ qui est le champ intérieur. Nous pouvons alors écrire [Durand 66] :

$$\varepsilon_0(\vec{E}_{(+)} - \vec{E}_{(-)}) = \varepsilon_0(E_{(+)} - E_{(-)}).\vec{n} = \sigma.\vec{n} \tag{20}$$

Où σ est une constante de dimension d'une densité surfacique de charges [C/m²]. Rappelons nous que nous sommes à l'équilibre électrostatique, c'est-à-dire que $\vec{E}_{(-)} = 0$. Nous pouvons écrire :

$$\sigma = -\varepsilon_0 \vec{\nabla} V_{(+)}.\vec{n} = -\varepsilon_0 \frac{\partial V_{(+)}}{\partial n} \tag{21}$$

Où $V_{(+)}$ est le potentiel du coté extérieur au conducteur. Notons que dans (21), au voisinage de la surface le gradient (à gauche) est équivalent à la dérivée partielle (à droite) car le champ électrique est normal au voisinage immédiat de la surface [Durand 66].

Le potentiel est continu sur la surface du conducteur :

$$V_{(+)} = V_{(-)} = V \tag{22}$$

Où $V_{(-)}$ est le potentiel du coté intérieur. L'équation intégrale à résoudre s'écrit finalement :

$$V = \frac{1}{4\pi\varepsilon_0} \iint_S \frac{\sigma}{r} dS \tag{23}$$

Cette formulation relie le potentiel à la surface du conducteur à la densité surfacique de charges qui est notre inconnue.

3.2 Système d'équations linéaires

Nous allons construire un système d'équations linéaires qui nous permettra de résoudre numériquement l'équation intégrale. Ce système d'équations sera mis sous forme matricielle, cette matrice est appelée matrice d'interaction. La première étape sera de discrétiser la géométrie en éléments finis, puis nous discuterons des choix de l'ordre des éléments et de la technique de projection qui auront des conséquences en terme de précision et de complexité de calcul.

3.2.1 Discrétisation de l'équation intégrale

Les géométries étant souvent complexes, il n'est pas possible de résoudre l'équation intégrale analytiquement. Les surfaces sont alors maillées en éléments surfaciques plus simples :

$$S = \sum_{j=1}^{N_e} S_j \tag{24}$$

Avec N_e le nombre d'éléments du maillage. L'équation intégrale discrétisée devient :

$$V = \frac{1}{4\pi\varepsilon_0} \sum_{j=1}^{N_e} \iint_{S_j} \frac{\sigma_j}{r_j} dS_j \tag{25}$$

3.2.2 Généralités sur les fonctions d'interpolations

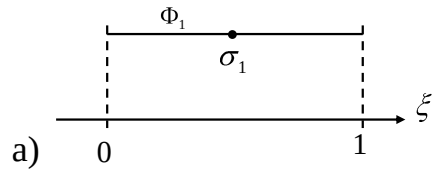
Des distributions surfaciques de charges sont associées à chaque élément, elles sont approximées par des interpolations polynomiales :

$$\sigma_j = \sum_{k=1}^{M_j} \sigma_k^j \, \Phi_k^j \tag{26}$$

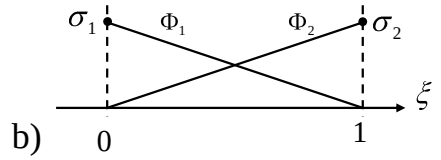
Où les Φ_k^j sont les fonctions d'interpolations (ou fonctions de forme) de l'élément *j*, les cœfficients σ_k^j sont nos inconnues associées à l'élément *j*, et M_j est le nombre de fonctions

de forme de l'élément *j*. Par soucis de lisibilité, nous n'écrirons plus tous les exposants *j* dans la suite des développements de la formulation.

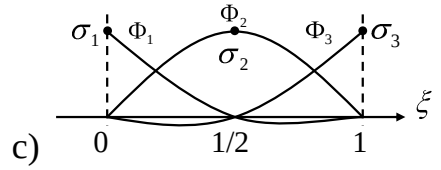
La Figure 8 présente quelques fonctions d'interpolations sur un élément de référence linéique. Notons que pour des raisons de vitesses de calcul, les polynômes d'interpolations sont prédéterminés sur des éléments de références. Ils sont ensuite projetés sur l'élément réel via une transformation géométrique. A l'ordre 0, les éléments sont uniformément chargés (Figure 8, a). Le nombre d'inconnues est alors le nombre d'éléments du maillage. A l'ordre 1 (Figure 8, b) les inconnues sont situées sur les nœuds, la densité de charge est une interpolation linéaire sur les éléments. Le nombre d'inconnues est ici le nombre de nœuds du maillage. A l'ordre 2 (Figure 8, c) les interpolations sont quadratiques. Le nombre d'inconnues est la somme du nombre d'éléments et du nombre de nœuds du maillage (pour des éléments linéiques).



$$\Phi_1(\xi) = 1$$
$$\sigma(\xi) = \sigma_1 \Phi_1$$



$$\Phi_1(\xi) = 1 - \xi,\ \Phi_2(\xi) = \xi$$
$$\sigma(\xi) = \sigma_1 \Phi_1 + \sigma_2 \Phi_2$$



$$\Phi_1(\xi) = (2\xi - 1)(\xi - 1),\ \Phi_2(\xi) = 4\xi(1 - \xi),$$
$$\Phi_3(\xi) = 4(2\xi - 1)$$
$$\sigma(\xi) = \sigma_1 \Phi_1 + \sigma_2 \Phi_2 + \sigma_3 \Phi_3$$

Figure 8 : Fonctions d'interpolations sur un élément de référence linéique en 1D : a) ordre 0, b) ordre 1 et c) ordre 2.

L'équation intégrale s'écrit :

$$V = \frac{1}{4\pi\varepsilon_0} \sum_{j=1}^{N_e} \sum_{k=1}^{M_j} \sigma_k^j \iint_{S_k} \frac{\Phi_k}{r_k} dS_k \qquad (27)$$

Une fois les inconnues projetées sur le maillage, nous choisissons une méthode de projection de l'équation intégrale comme par exemple la méthode de collocation ou de Galerkin.

3.2.3 Méthode de collocation

La méthode de collocation (ou corrélation par points) vérifie l'équation intégrale sur certains points. Pour *N* inconnues σ_i il nous faut écrire *N* équations :

$$V_i = \frac{1}{4\pi\varepsilon_0} \sum_{j=1}^{N_e} \sum_{k=1}^{M_j} \sigma_k^j \iint_{S_k} \frac{\Phi_k}{r_{ik}} dS_k \tag{28}$$

Ordre 0

A l'ordre 0 les charges sont supposées constantes sur chaque élément. Les équations intégrales sont vérifiées sur les barycentres des éléments :

$$V_i = \frac{1}{4\pi\varepsilon_0} \sum_{j}^{N_e} \sigma_j \iint_{S_j} \frac{1}{r_{ij}} dS_j \tag{29}$$

Nous obtenons alors un système d'équations linéaires de N_e inconnues :

$$G_{ij}\sigma_j = V_i \tag{30}$$

Avec :

$$G_{ij} = \frac{1}{4\pi\varepsilon_0} \iint_{S_j} \frac{1}{r_{ij}} dS_j \tag{31}$$

L'ensemble des G_{ij} forme la matrice d'interaction. La matrice est entièrement pleine et n'est pas symétrique. Elle reste cependant assez bien conditionnée car les termes diagonaux sont dominants (décroissance en 1/r autour de la diagonale) [Krahenbuhl 83]. La résolution de ce système d'équations linéaires donnera les N_e densités de charge surfaciques constantes par éléments. N_e x N_e x N_G calculs sont nécessaires, avec N_G le nombre de points de Gauss nécessaire à l'intégration numérique (voir paragraphe 3.3.1).

Ordre 1

A l'ordre 1 les équations sont vérifiées aux nœuds des éléments, nous avons alors *Nd* inconnues avec *Nd* le nombre de nœuds du maillage. La construction de la matrice demandera *Ne* x *Nd* x M_j x N_G calculs. Les cœfficients associés aux inconnues de chaque élément sont projetés dans le système d'équation global, c'est l'assemblage. Il est ici plus complexe qu'à l'ordre 0 car il faut établir des relations de correspondances entre les inconnues locales par élément et celles globales au maillage. Il rend également la parallélisation plus compliquée.

Une remarque, notre formulation intégrale peut engendrer des effets de pointes sur les géométries contenants des coins. Or à l'ordre 1 en collocation, nous vérifions justement les équations en ces points divergents ! Cette formulation ne semble pas adaptée, nous verrons dans la partie application (4.3.1) que nous obtenons de grandes imprécisions sur le calcul des capacités.

3.2.4 Méthode de Galerkin

Nous proposons de projeter l'équation intégrale (27) sur un ensemble de fonctions test. Nous choisissons d'utiliser les fonctions de forme utilisées pour interpoler les solutions : c'est la méthode de Galerkin. Nous pouvons alors écrire pour N fonctions :

$$\iint_S \left(\frac{1}{4\pi\varepsilon_0} \sum_{j=1}^{N_e} \sum_{k=1}^{M_j} \sigma_k^j \iint_{S_k} \frac{\Phi_k}{r_k} dS_k - V \right) . \Phi_i \, dS = 0 \tag{32}$$

Où Φ_i est l'ensemble des fonctions de forme de l'élément *i*. En développant Φ_i nous obtenons :

$$\sum_{l=1}^{M_i} \iint_{S_l} \left(\frac{1}{4\pi\varepsilon_0} \sum_{j=1}^{N_e} \sum_{k=1}^{M_j} \sigma_k^j \iint_{S_k} \frac{\Phi_k}{r_{lk}} dS_k - V_i^l \right) . \Phi_l \, dS_l = 0 \tag{33}$$

Avec M_i le nombre de fonctions de forme de l'élément *i*. Nous pouvons réécrire les équations sous la forme suivante :

$$\frac{1}{4\pi\varepsilon_0} \sum_{j=1}^{N_e} \sum_{k=1}^{M_j} \sum_{l=1}^{M_i} \sigma_k^j \iint_{S_k} \iint_{S_l} \frac{\Phi_k \Phi_l}{r_{kl}} dS_k \, dS_l = \sum_{l=1}^{M_i} \iint_{S_l} V_l^i \, \Phi_l \, dS_l \tag{34}$$

Ordre 0

Les fonctions de formes sont constantes. L'intégration par la méthode de Galerkin à l'ordre 0 est alors une simple double intégration, une sur l'élément où le potentiel est calculé et une sur l'élément contenant la source des charges. Le système d'équations linéaire de N_e inconnues s'écrit alors :

$$\frac{1}{4\pi\varepsilon_0}\iint_{S_i}\left[\sigma_j \iint_{S_j}\frac{1}{r_{ij}}dS_j\right]dS_i = \iint_{S_i} V_i \; dS_i \tag{35}$$

Ce système d'équations linéaires peut se mettre sous la forme :

$$\tilde{G}_{ij}.\sigma_j = V_i.S_i \tag{36}$$

Les coefficients de la matrice d'interaction $\tilde{G}_{ij}$ sont alors :

$$\tilde{G}_{ij} = \frac{1}{4\pi\varepsilon_0}\iint_{S_i}\left[\iint_{S_j}\frac{1}{r_{ij}}dS_j\right]dS_i \tag{37}$$

Tout comme pour la méthode de collocation, la matrice d'interaction est pleine, cependant elle est symétrique. La méthode de Galerkin est plus précise que la méthode de collocation car les équations intégrales sont vérifiées en tout point de la surface de l'élément et pas uniquement en son barycentre. Cependant la complexité du calcul est plus élevée, elle est ici de N_e x N_e x N_G x N_G soit N_G fois plus importante que celle de la collocation. La matrice étant symétrique il est possible de n'en calculer que la moitié et de réduire ainsi la complexité d'un facteur 2.

Ordre 1

La méthode de Galerkin à l'ordre 1 fait preuve en théorie d'une grande précision, cependant son coût est très élevé : N_e x N_e x M_i x M_j x N_G x N_G opérations sont nécessaires au calcul de la matrice d'interaction. Par exemple avec un maillage triangulaire (3 fonctions de forme par élément) le calcul de la matrice d'interaction est 9 fois plus long qu'à l'ordre 0.

3.3 Choix de la méthode d'intégration

Nous avons deux méthodes à notre disposition pour calculer les intégrales nécessaires à la construction de la matrice d'interaction : la méthode numérique d'intégration par points de Gauss et l'utilisation de solutions analytiques. Les calculs sont couramment effectués avec la méthode numérique des points de Gauss car elle est très rapide, cependant cette méthode peut s'avérer imprécise voire conduire à des singularités artificielles. L'alternative est l'utilisation de solutions analytiques, cette approche est plus précise mais également plus coûteuse en temps de calcul. La solution optimale se trouve peut-être en l'hybridation des approches numériques et analytiques du calcul d'intégrales, ainsi le meilleur ratio entre précision et temps de calcul, en d'autres mots la performance, devrait être obtenue. Nous allons présenter dans cette partie la méthode d'intégration par points de Gauss ainsi que les critères d'utilisation des solutions analytiques.

3.3.1 Intégration numérique par points de Gauss

Cette méthode est appréciée pour sa vitesse de calcul élevée car seule l'évaluation de la fonction à intégrer sur quelques points prédéfinis est nécessaire :

$$\int f(x)\,dx \approx \sum_i f(x_i)\,w_i \tag{38}$$

Où les x_i sont les points de Gauss et w_i les poids de Gauss. La méthode la plus couramment employée est la méthode de Gauss-Legendre. Les points de Gauss sont alors les racines des polynômes de Legendre [Weisstein 2005].

3.3.2 Corrections analytiques

Dans les problèmes 3D que nous traitons, les surfaces des conducteurs sont généralement maillées en triangles. Il est alors utile d'avoir une solution de l'intégrale (14) pour ce type d'éléments. Une méthode analytique d'évaluation du potentiel et du champ électrique créés par des triangles a été développée [Wilton 79, Graglia 93, Hachi-Ashtiani 98, Janssen 10, Rubeck 11b], les détails de la méthode sont présentés dans l'Annexe B.

Lorsque l'intégration numérique est imprécise ou singulière, nous utiliserons ces solutions analytiques. L'avantage des solutions analytiques est la grande précision de l'évaluation de l'intégrale. Cependant les formulations mathématiques sont souvent

complexes ce qui nuit à la vitesse de calcul. Ces calculs sont difficilement vectorisables en général. Nous utilisons alors les solutions analytiques de manière ponctuelle, nous présentons dans cette partie les critères d'utilisation de ces solutions.

Méthode de collocation

Les imprécisions (voire les singularités) se présentent lors de l'interaction d'un élément sur lui-même, c'est-à-dire lorsque $i=j$ dans l'équation (28).

A l'ordre 0 elles sont localisées sur les éléments diagonaux de la matrice d'intégration. Les points de Gauss peuvent se retrouver très proches voire confondus avec le point de collocation. La conséquence est que l'intégrale est surévaluée, donc imprécise, voire singulière (division par zéro). La diagonale de la matrice est donc calculée avec les solutions analytiques.

A l'ordre 1 le calcul numérique n'est pas singulier, mais il est toujours imprécis. Les solutions analytiques sont utilisées pour le calcul des intégrales sur les fonctions de forme d'un élément lorsque le point d'intégration appartient à ce même élément.

Méthode de Galerkin

A l'instar de la collocation, le problème se situe lors de l'intégration d'un élément sur lui-même. L'équation (34) présente une double intégration surfacique. La double intégration par points de Gauss s'écrit sous la forme :

$$\iint_S \iint_{S'} \frac{1}{r}\, dS'\, dS \approx \sum_i \sum_j \frac{1}{r_{ij}}\, w_i\, w_j \tag{39}$$

Pour éviter les imprécisions et les singularités, nous effectuons une des intégrations analytiquement :

$$\iint_S \iint_{S'} \frac{1}{r} dS'\, dS \approx \sum_i \left(\iint_{S'} \frac{1}{r_i} dS' \right) w_i \tag{40}$$

3.4 Calcul des capacités

Nous avons établi plusieurs formulations qui lient le potentiel sur la surface d'un conducteur à la densité surfacique de charges. La résolution du système d'équations nous donne les densités de charges à la surface des éléments du maillage. Nous pouvons calculer

les capacités entre les conducteurs à partir de ces densités de charges. L'ensemble des capacités entre tous les conducteurs forme la matrice de capacités [Ardon 10b, Aimé 09]. Elle est de dimension $n_x n$ avec n le nombre de régions conductrices. Il existe deux définitions des capacités : les capacités de Maxwell et les capacités de Kirchhoff.

3.4.1 Charge d'un élément de conducteur

La charge totale q_i [C] d'un élément i d'un conducteur se calcule à partir de la densité surfacique de charges. Nous pouvons écrire :

$$q_i = \iint_{S_i} \sigma_i \, dS_i \tag{41}$$

Nous rappelons que les densités de charges sont interpolées par des polynômes, le calcul de la charge devient :

$$q_i = \sum_k \sigma_k^i \iint_{S_k} \Phi_k \, dS_k \tag{42}$$

3.4.2 Capacités de Maxwell

La capacité C_{ij} [F] entre les conducteurs i et j est donnée par :

$$C_{ij} = \frac{Q_j^{(i)}}{V_i - V_{ref}} \tag{43}$$

Où $Q_j^{(i)}$ est la charge totale [C] du conducteur j obtenue avec le conducteur i soumis à V_i. V_i est choisi à 1V et tous les autres conducteurs sont soumis à 0V. Le potentiel de référence V_{ref} se situe à l'infini et il est supposé nul. Nous pouvons écrire les termes de la matrice de capacité de la manière suivante [Aimé 09] :

$$C_{ij} = \sum_{k=1}^{N(j)} \varepsilon_{rk}^{(j)} \, q_k^{(j)} \tag{44}$$

Où $N(j)$ est le nombre d'élément de la région j, q_k est la charge de l'élément k et ε_{rk} est la permittivité diélectrique relative du milieu environnant à l'élément k. Elle vaut 1 si l'élément est en contact avec du vide ou de l'air et la moyenne des permittivités si l'élément est en contact avec deux milieux. Les capacités propres C_{ii} sont positives, les

capacités mutuelles C_{ij} sont négatives. Ces capacités sont appelées capacités de Maxwell. En théorie la matrice des capacités et symétrique.

3.4.3 Capacités de Kirchhoff

Les capacités de Maxwell ne sont pas les capacités réelles et elles ne peuvent pas être utilisées dans les circuits électriques. Les capacités au sens de Kirchhoff sont la combinaison des capacités mutuelles avec des capacités propres. Les capacités de Kirchhoff sont obtenues à partir des capacités de Maxwell par la relation :

$$\begin{cases} C'_{ii} = \sum_{j=1}^{n} C_{ij}, & \text{pour tout i} \\ C'_{ij} = -C_{ij}, & \text{si i} \neq \text{j} \end{cases} \tag{45}$$

La capacité propre de Kirchhoff C'_{ii} représente la capacité entre le conducteur *i* soumis au potentiel V_i et l'infini de potentiel 0V. La capacité mutuelle C'_{ij} est positive et elle représente la capacité entre les conducteurs *i* et *j* soumis respectivement aux potentiels V_i et V_j. La matrice des capacités de Kirchhoff est également symétrique.

3.5 Choix d'un solveur

Les potentiels aux conducteurs sont fixés en fonction des capacités que nous souhaitons calculer, il nous faut à présent résoudre le système d'équations linéaires afin d'obtenir les densités de charges surfaciques sur les conducteurs.

Il existe deux grandes familles de méthodes de résolution des systèmes linéaires : les solveurs directs et les solveurs itératifs. Les matrices sont ici pleines, une résolution directe telle qu'une décomposition LU est de complexité $O(N^3)$ avec N la dimension de la matrice carrée. Une telle complexité demande énormément de temps de calcul. Un dernier point, l'utilisation d'un solveur direct requiert la connaissance de la totalité de la matrice d'interaction. C'est bien le cas ici mais à partir du Chapitre IV la matrice sera approximée par morceaux ce qui exclura l'utilisation d'un solveur direct.

Nous choisissons donc d'utiliser un solveur itératif pour la résolution du système d'équations linéaires. L'idée est d'approcher graduellement la solution réelle par une succession de solutions approximatives. Nous parlons de convergence lorsque la solution

s'approche de la solution réelle et de divergence lorsqu'elle s'en éloigne. Les algorithmes que nous utilisons reposent sur l'évaluation de produits matrice-vecteur. Ils sont de complexité $O(N^2)$ car les matrices sont pleines. Ces méthodes semblent donc plus intéressantes car plus rapides que les solveurs directs, de plus le produit matrice-vecteur est facilement parallélisable. Cependant, ces méthodes ne sont pas toujours capables de trouver une solution, l'utilisation de préconditionneurs plus ou moins sophistiqués pour améliorer le conditionnement de la matrice est alors nécessaire.

4 Performance de la formulation

4.1 Description du cas test

Soit un condensateur plan constitué de deux plaques conductrices en vis-à-vis (Figure 9), nous proposons de calculer la capacité mutuelle entre les deux plaques. Les dimensions des plaques sont 10x10 mm² et elles sont espacées de 2 mm. Une plaque est polarisée à 1V et l'autre à 0V. Les plaques sont maillées en triangles, le nombre d'élément varie entre 500 et 16000. Nous comparons la précision en fonction de la méthode d'intégration et de l'ordre des formulations. Nous comparons également les performances des méthodes de résolution.

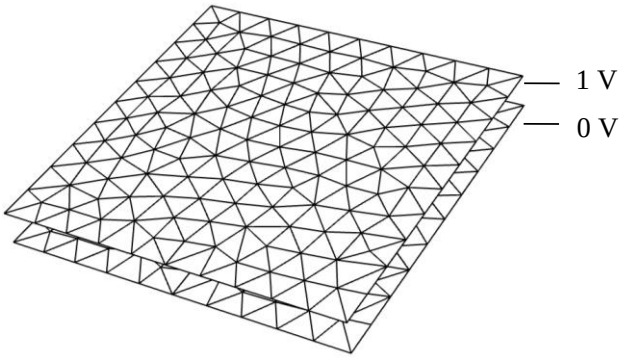


Figure 9 : Condensateur plan maillé en triangles.

4.2 Solveur itératif

Les solveurs itératifs linéaires les plus courants approchent la solution réelle par des méthodes qui minimisent le résidu (Figure 10). La méthode générale est la suivante : le vecteur solution (ici x) est tout d'abord initialisé, souvent à zéro. Le résidu peut alors être

calculé depuis cette solution. Le résidu est noté r, A est la matrice du système d'équations et b est le second membre. Une nouvelle solution est alors déterminée. La principale différence entre les différents algorithmes réside en la méthode de calcul de la solution suivante à partir du résidu. Le calcul s'arrête lorsque la solution satisfait le critère de convergence. Le grand intérêt de ces méthodes est qu'elles peuvent voir la matrice comme une boite noire, seule une méthode de calcul du résidu est nécessaire à la mise en œuvre de la résolution.

```
x = x0 // Initialisation de la solution
Tant que (critère de convergence)
   r = A * x - b // Calcul du résidu
   x = f( r ) // Détermination d'une nouvelle solution
Fin
```

Figure 10 : Schéma de principe d'un algorithme de résolution itérative.

La méthode du gradient conjugué (GC) est une méthode itérative très connue [Hestenes 52]. La minimalisation du résidu est effectuée par une descente de gradient. C'est une méthode adaptée aux matrices symétriques, ce qui est le cas avec l'intégration de Galerkin et quasiment le cas pour la collocation (les valeurs des éléments symétriques de la matrice sont du même ordre de grandeur). Pour les matrices quelconques il existe une variante, le gradient biconjugué qui calcule également des résidus sur la transposée de la matrice [Fletcher 76].

La méthode de résolution implémentée dans nos codes est GMRES pour Generalized Minimal Residual Method [Saad 86]. Cette méthode est adaptée à la résolution de systèmes d'équations linéaires non symétriques. Son principe de fonctionnement repose sur l'approximation de la solution dans un sous espace de Krylov en minimisant le résidu.

Nous présentons en Figure 11 une comparaison des temps de résolution entre une décomposition LU et GMRES (critère de convergence à 1e-12). L'intégration est effectuée en collocation à l'ordre 0. La complexité en $O(N^3)$ de la résolution LU est bien visible. Il est impensable d'utiliser cette méthode avec un nombre d'éléments de l'ordre de la dizaine ou de la centaine de millier.

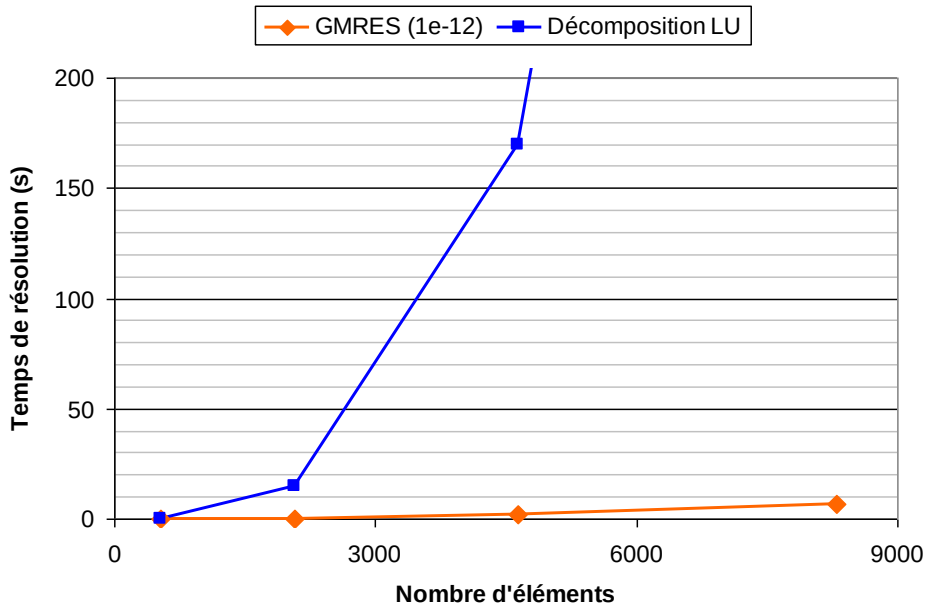


Figure 11 : Comparaison des temps de résolution entre un solveur direct LU et un solveur itératif GMRES en fonction du nombre d'éléments. Intégration par collocation à l'ordre zéro.

4.3 Comparaison des méthodes d'intégration et des ordres des formulations

La matrice d'interaction est construite de quatre méthodes différentes : à l'ordre 0 et à l'ordre 1, et pour chacun de ces cas l'intégration est effectuée par la méthode de collocation et par la méthode de Galerkin. La Figure 12 montre la répartition de la distribution surfacique de charges sur le condensateur plan à l'ordre 0 et à l'ordre 1, les éléments sont uniformément chargés à l'ordre 0, tandis qu'à l'ordre 1 les charges sont des fonctions polynomiales du premier degré.

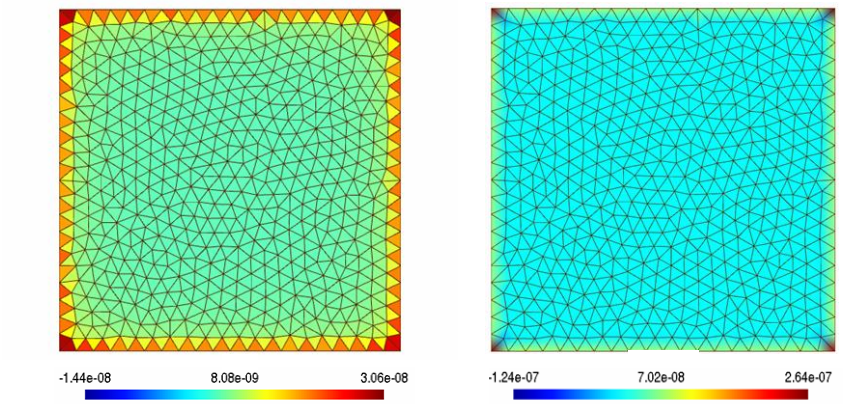


Figure 12 : Distribution surfacique de charges à la surface d'un condensateur plan de 2000 éléments triangulaires. Intégration par points de collocations, à gauche ordre 0, à droite ordre 1.

4.3.1 Précision des formulations

Nous comparons à la Figure 13 la capacité du condensateur plan en fonction du maillage pour différentes méthodes d'intégration et ordre des éléments. La référence est l'asymptote vers laquelle toutes les courbes semblent converger. Le calcul des intégrales est effectué par points de Gauss, les singularités artificielles sont corrigées par des solutions analytiques.

La méthode de collocation à l'ordre 1 ne permet pas une bonne évaluation de la distribution en charges, en effet rappelons-nous que les intégrales sont calculées sur les nœuds et que des effets de pointes sont localisés sur les bords. La collocation en ces points particuliers induit une erreur conséquente. A nombre d'éléments constant, la méthode la plus précise est l'intégration de Galerkin à l'ordre 1. La méthode la moins précise est la collocation à l'ordre 0. Ces résultats sont en accord avec ce qui a été discuté lors des présentations des formulations.

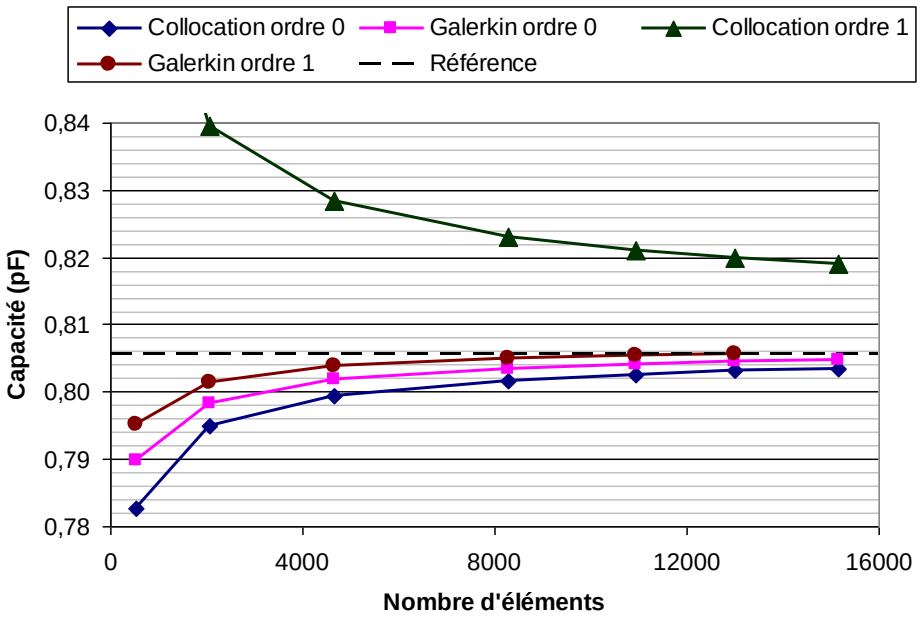


Figure 13 : Comparaison de la valeur de la capacité du condensateur plan en fonction du nombre d'éléments pour différentes méthodes d'intégration.

4.3.2 Temps de construction de la matrice d'interaction

Nous présentons à la Figure 14 les temps d'intégration en fonction du nombre d'éléments. Nous comparons les résultats pour les quatre méthodes d'intégration. Nous

avons vu au point précédent que la méthode de Galerkin à l'ordre 1 est la méthode la plus précise, mais c'est également la plus coûteuse ; elle est d'un ordre de grandeur plus lente que la méthode de Galerkin à l'ordre 0 et de deux ordres de grandeur plus lente que la méthode de collocation à l'ordre 0. Ces résultats reflètent bien les complexités des formulations discutées précédemment (3.2.4).

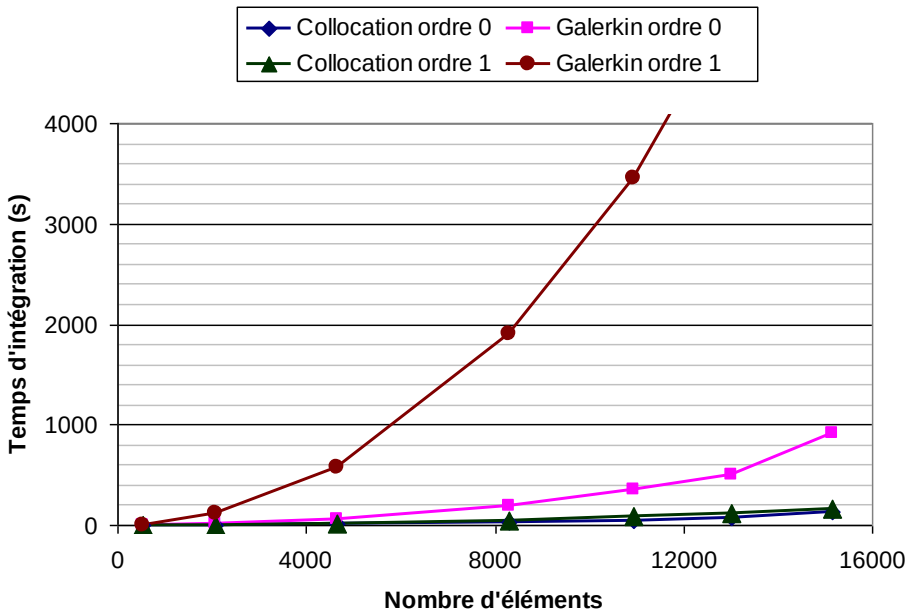


Figure 14 : Temps d'intégration en fonction du nombre d'éléments pour différentes méthodes d'intégration.

4.3.3 Temps de résolution du système d'équations

Les temps de résolution sont présentés à la Figure 15. Les résolutions des formulations à l'ordre 1 sont plus rapides que celles à l'ordre 0. L'explication est que le nombre de degrés de liberté est moindre à l'ordre 1 (inconnues projetées sur les nœuds du maillage) qu'à l'ordre 0 (inconnues projetées sur les éléments). Nous constatons également que pour un ordre donné, les méthodes de Galerkin convergent plus rapidement que celles par collocation. L'explication ici est que les matrices d'interactions issues de la méthode de Galerkin sont mieux conditionnées, à l'ordre 0 par exemple elles sont symétriques contrairement à celles obtenues par la méthode de collocation.

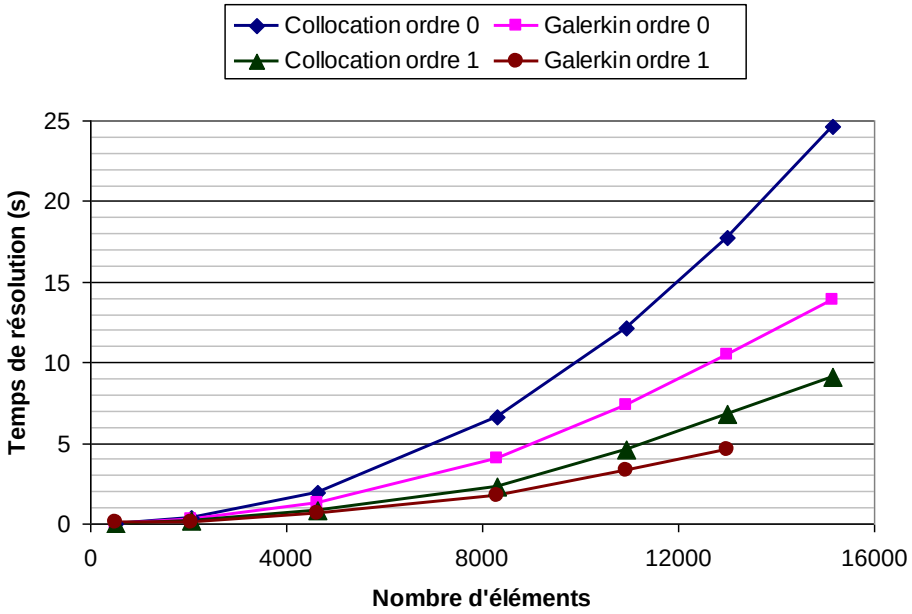


Figure 15 : Temps de résolution en fonction du nombre d'éléments pour différentes méthodes d'intégration.

4.3.4 Performance des formulations

Récapitulons ce qui a été montré précédemment, plus un calcul est complexe et plus il est précis, mais il est également plus coûteux en temps de calcul. Quelle formulation apporte le meilleur rapport entre précision et temps de calcul ?

La Figure 16 montre les temps d'intégration en fonction de l'erreur relative sur la capacité pour les différentes formulations. Pour une erreur donnée nous voyons que la méthode de collocation à l'ordre 0 est la plus rapide. Par conséquent, il semble qu'il soit plus intéressant d'un point de vue des performances de raffiner le maillage plutôt que de monter en complexité dans la formulation. Notons cependant qu'une augmentation du nombre d'éléments nécessitera une quantité de mémoire vive plus importante.

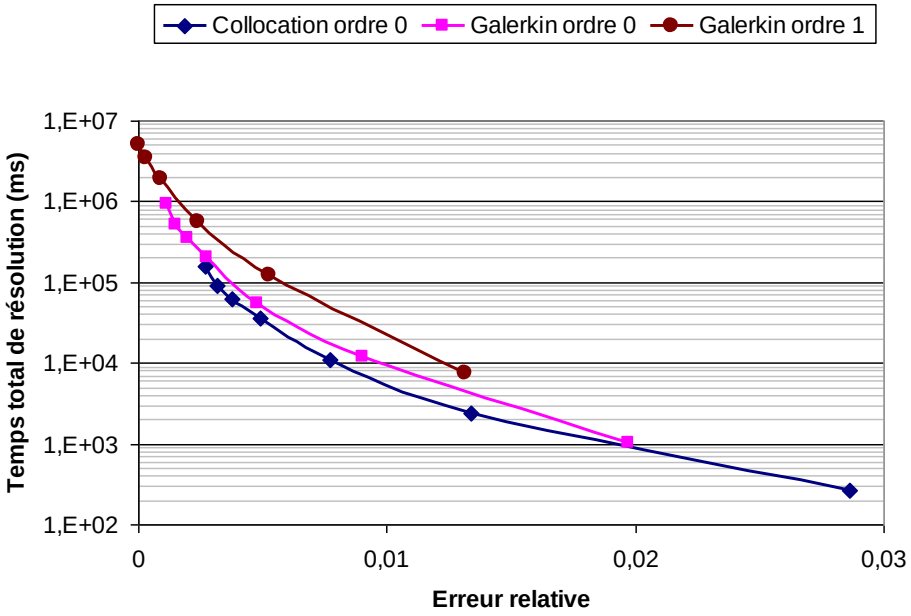


Figure 16 : Temps total de résolution (intégration+résolution) en fonction de l'erreur relative sur la capacité pour différentes formulations.

4.4 Conclusion sur le choix de la formulation

Nous avons comparé les performances de différentes implémentations de notre formulation intégrale électrostatique. Les formulations en ordre 1 sont plus précises que celles en ordre 0, cependant elles sont bien plus coûteuses d'un point de vu calculatoire. De manière analogue, l'intégration par la méthode de Galerkin est plus précise que celle par points de collocation, mais est aussi plus coûteuse. Il est plus intéressant d'augmenter le nombre de mailles plutôt que la complexité de l'assemblage, à condition néanmoins d'avoir les ressources nécessaires en mémoire vive. De plus, les formulations à l'ordre 0 sont massivement parallélisables, en effet chaque terme de la matrice est calculé de manière totalement indépendante. Par conséquent dans la suite du mémoire nous ne travaillerons plus que sur la formulation en potentiel électrostatique développée en collocation à l'ordre 0.

5 Vectorisation de la formulation intégrale

La vectorisation permet d'optimiser les accès à la mémoire et d'exploiter les pipelines des processeurs. Les données sont traitées continûment par paquets, c'est-à-dire que nous travaillons sur des tableaux et non plus sur des variables.

5.1 Description du cas test

Le cas test que nous utilisons ici est le calcul de la capacité propre d'une sphère.

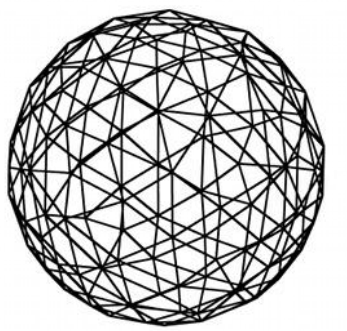


Figure 17 : Sphère maillée en triangles, une solution analytique exacte de la capacité électrostatique existe pour cette géométrie.

L'avantage ce cette géométrie est qu'il existe une solution analytique exacte, ce qui est très intéressant pour évaluer la précision des modèles [Rjasanow 07]. De plus, l'absence d'effet de bords sur une sphère assurera une bonne stabilité de la solution lorsque nous allons augmenter le nombre de degrés de liberté. Les sphères sont maillées en triangles (Figure 17), éléments géométriques de base sur lequel nous avons des solutions analytiques pour le potentiel et le champ électrique.

La capacité propre d'une sphère polarisée uniformément est donnée par :

$$C = 4\pi\varepsilon_0 R \tag{46}$$

Où R est le rayon de la sphère [m]. La distribution de charge à la surface de la sphère est homogène, elle est donnée par :

$$\sigma = \varepsilon_0 \frac{V_0}{R} \tag{47}$$

La précision des modèles sera évaluée sur la capacité globale, ainsi que sur les distributions de charges locales. En effet il est possible que la capacité globale soit exacte mais que la distribution de charges quant à elle soit largement inhomogène.

5.2 Approche vectorisée du calcul en Java

Le calcul de la matrice d'interaction est de complexité N². Il est nécessaire d'être efficace dans son calcul. La première étape afin d'éviter des calculs redondants est la génération des tables de points de Gauss (Figure 18), ainsi que les poids de Gauss et les déterminants des jacobiens, sans oublier également la table des points de collocation [Peng 08].

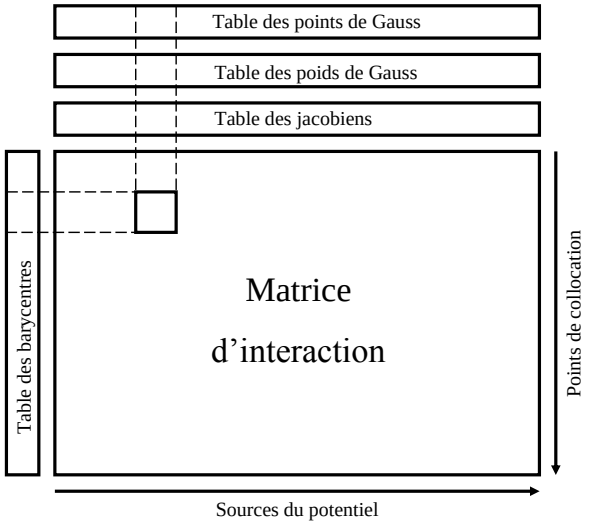


Figure 18 : Structure de la matrice d'interaction et des tables nécessaires à son calcul.

Classiquement, l'algorithme possède trois boucles imbriquées (Figure 19) : la première sur les points de collocation, et la deuxième sur les éléments sources de potentiels. L'intégration numérique par points de Gauss nécessite une troisième boucle. Notons ici que la boucle 3 itère sur très peu de données (moins d'une dizaine en général), le pipelining n'est pas exploité au maximum.

```
// Boucle 1: points de collocation
For i = 0,1,…N
   // Boucle 2: sources de potentiel
   For j = 0,1,…N
      // Boucle 3: intégration de Gauss
      For k = 0,…nombre de points de Gauss
         V(i,j) += 1/r(i,j,k) * poids(k) * jacobien(k)
      End
   End
End
```

Figure 19 : Algorithme classique de construction de la matrice d'interaction.

Le calcul performant de la matrice d'interaction est obtenu par la vectorisation. Concrètement, à l'instar de l'algorithme de multiplication matricielle présenté au premier chapitre (2.4.4), les boucles 2 et 3 vont être interverties. Il est possible de le faire car le calcul de l'intégrale est indépendant de l'ordre des boucles. De plus des opérateurs vectoriels sont utilisés, le calcul de la matrice se fait alors ligne par ligne (Figure 20).

Une classe performante Java de manipulation matricielle a été développée au laboratoire par Jean-Louis Coulomb [Coulomb 12]. Elle repose sur une double indexation sur les lignes et sur les colonnes (voir chapitre I, 2.4.2) où toutes les opérations arithmétiques sont vectorisées.

```
// Boucle 1: points de collocation
For i = 0,1,…N
   // Boucle 2: intégration de Gauss
   For k = 0,1,… nombre de points de Gauss
      // Boucle 3: sources de potentiel
      For j = 0,1,…N
         V(i,j) += 1/r(i,j,k) * poids(k) * jacobien(k)
      End
   End
End
```

Figure 20 : Calcul vectorisé de la matrice d'interaction. Les boucles 2 et 3 ont été interverties afin de mieux exploiter le pipelining.

Une fois la matrice calculée, sa diagonale (intégration des éléments sur eux-mêmes) est remplacée par des solutions analytiques car l'intégration de Gauss introduit une singularité artificielle ici. L'étude de l'influence de la qualité de la diagonale sur la précision des résultats de nos problèmes sera présentée au paragraphe suivant.

Une comparaison des performances entre l'algorithme classique et sa version vectorisée est présentée à la Figure 21. Les calculs sont effectués avec 7 points de Gauss et une diagonale analytique. Le problème est le calcul de la capacité d'une sphère maillée en triangle. Les gains sont de 46% dans le cas à 11.000 éléments et 39% dans le cas à 20.000 éléments. Ces gains apportés par la vectorisation sont appréciables compte tenu du peu d'effort de la part du développeur pour optimiser l'algorithme (une simple inversion de boucles).

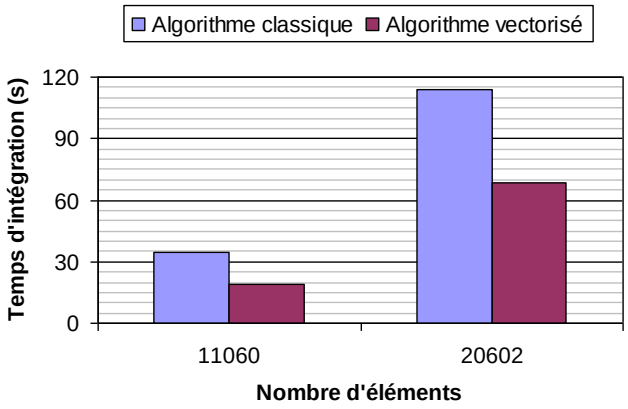


Figure 21 : Comparaison sur des temps d'intégration entre l'algorithme classique et sa version vectorisée. Les gains sont de 46% et 39% pour respectivement le premier et le second cas.

5.3 Hybridation intégration numérique et analytique

Nous proposons ici une étude de l'influence de la qualité de l'intégration, notamment de la diagonale, sur le calcul de la distribution de charges. L'erreur est définie par la norme du vecteur qui fait la différence entre la distribution de charges calculée et la distribution théorique, ainsi les inhomogénéités non physiques sont détectées.

La Figure 22 présente l'erreur sur la distribution de charges en fonction du nombre d'éléments pour différentes techniques d'intégration. Dans les cas d'intégrations par points de Gauss, la diagonale de la matrice d'interaction est toujours corrigée avec des solutions analytiques. La première observation est que plus le nombre d'éléments augmente, plus l'erreur diminue. C'est un résultat attendu car plus la géométrie est discrétisée et plus cette discrétisation en triangles s'approche de la géométrie réelle, donc meilleure est la modélisation. Concernant la méthode d'intégration, l'intégration entièrement analytique de la matrice d'interaction est la méthode qui conduit à la distribution de charges de meilleure qualité. L'intégration à un seul point de Gauss est clairement insuffisante, l'erreur est presque d'un ordre de grandeur supérieure à celle générée par l'intégration purement analytique.

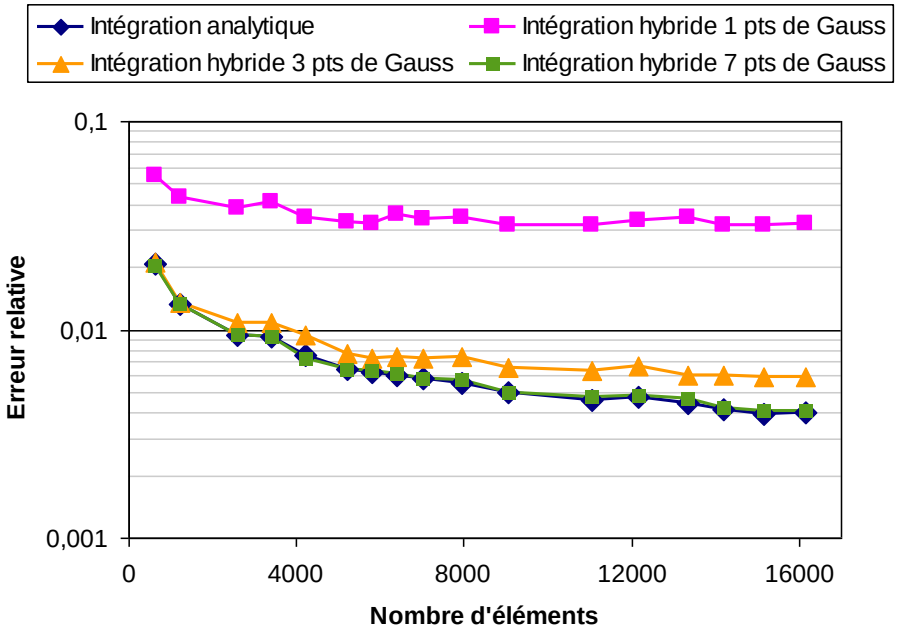


Figure 22 : Erreur relative de la distribution de charges en fonction du nombre d'éléments pour différentes méthodes d'intégration. L'erreur est la norme2 entre le vecteur solution et la valeur théorique de la distribution de charges. Les intégrations par points de Gauss sont toutes corrigées sur la diagonale par des solutions analytiques.

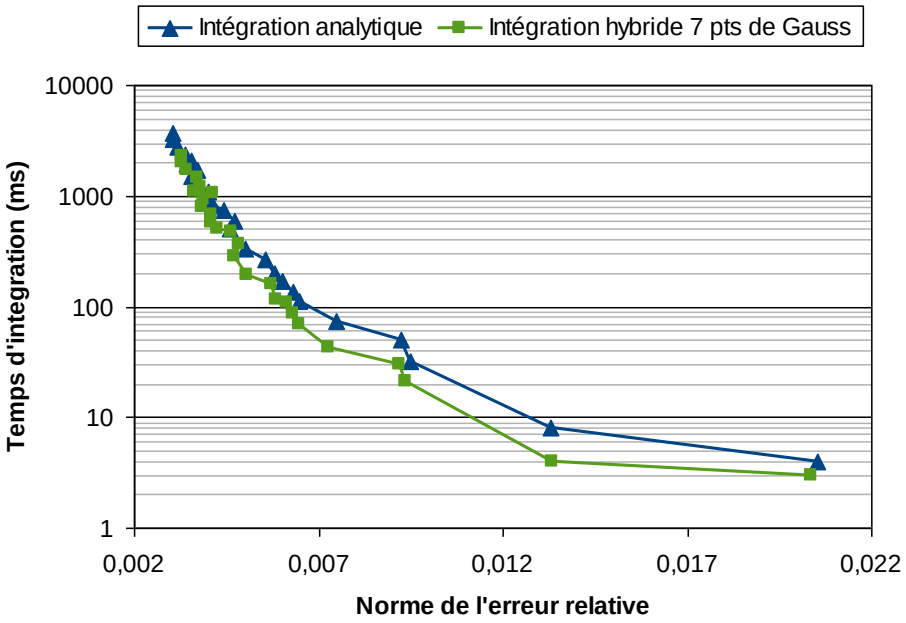


Figure 23 : Temps de construction de la matrice d'interaction en fonction de l'erreur pour une intégration purement analytique et une intégration hybride numérique à 7 points de Gauss avec des solutions analytiques sur la diagonale.

La précision sur la distribution de charges est bien meilleure que dans le cas à 1 points de Gauss, mais reste en deçà par rapport à l'intégration analytique. L'intégration numérique à 7 points de Gauss avec diagonale corrigée analytiquement semble être d'aussi bonne qualité que l'intégration purement analytique. Les erreurs sont ici quasiment équivalentes.

La précision de la distribution de charges entre une intégration analytique et une intégration numérique à 7 points de Gauss semble équivalente. Mais laquelle est la plus performante ? C'est-à-dire qui offre le meilleur ratio entre précision et temps de calcul. La Figure 23 présente les temps d'intégration en fonction de l'erreur. Nous observons que pour une erreur donnée, l'intégration par points de Gauss est plus rapide (x3 pour une erreur relative de 7e-3 par exemple), c'est par conséquent la méthode d'intégration la plus performante.

Nous trouvons dans la littérature quasi exclusivement des constructions de matrice de méthodes intégrales entièrement analytiques [Lezar 10], notre approche hybride combinant intégration analytique et numérique semble originale.

5.4 Simple précision versus double précision

Les architectures à base de cartes graphiques sont très efficaces, surtout en calcul en simple précision. Dans cette perspective, il est nécessaire de vérifier que les calculs de distributions de charges ne seront pas dégradés par la perte de précision sur les variables.

La Figure 24 présente l'erreur relative sur la distribution de charges en fonction du nombre d'éléments. Nous comparons trois formats de matrice d'interaction : en simple précision, en double précision et enfin en simple précision avec une diagonale en double. En effet, les intégrations en 1/r génèrent les valeurs de la matrice les plus fortes sur la diagonale (car r est très petit ici). Par conséquent, la qualité de la diagonale peut s'avérer déterminante sur la solution, c'est ce que nous cherchons à vérifier.

Les courbes sont parfaitement superposées, c'est-à-dire que pour une formulation en potentiel électrostatique, en collocation à l'ordre zéro, le calcul de la matrice d'interaction en simple précision est suffisant. Le calcul en simple précision n'introduisant pas plus d'erreur qu'en double, il peut être réalisé sur architecture GPGPU.

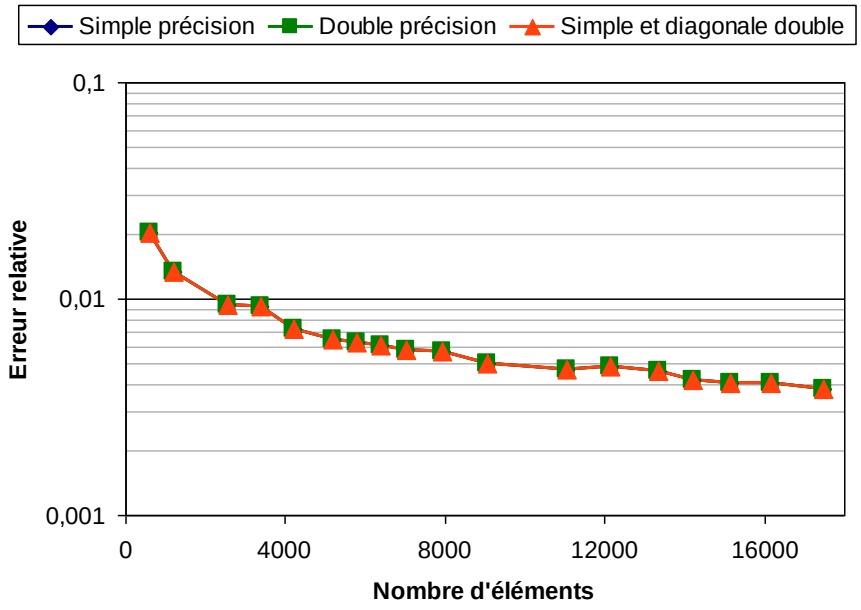


Figure 24 : Erreur sur la distribution en charges en fonction du nombre d'éléments pour différentes précisions de calcul de la matrice d'interaction.

6 Conclusion

Nous avons dans ce chapitre présenté une formulation intégrale très simple en potentiel électrostatique. Cette formulation servira d'exemple dans les chapitres suivants, toutefois les stratégies de parallélisation seront pour la plupart applicables à d'autres formulations. Nous avons développé cette formulation en collocation à l'ordre zéro, ce n'est pas la méthode la plus précise, mais c'est la plus rapide et celle qui se prête au mieux à une parallélisation massive sur architecture GPGPU.

L'étape précédant la parallélisation d'un code est son optimisation en mono processeur. En effet les gains apportés par l'optimisation d'un code peuvent être supérieur à ceux de sa parallélisation. L'optimisation la plus importante est la vectorisation car le pipelining est pleinement exploité. Cette optimisation n'est pas toujours difficile à mettre en œuvre, dans notre cas une simple inversion de boucles dans l'algorithme du calcul de la matrice d'interaction était requise pour des gains en vitesse de calcul appréciables (environ 40%).

L'intégration numérique par points de Gauss génère des singularités sur les termes diagonaux de la matrice d'interaction, c'est-à-dire lors de l'interaction d'un élément sur lui-même. Une solution analytique est alors utilisée dans ce cas précis afin d'éviter

l'apparition de singularités. Cette hybridation entre calcul numérique et analytique d'intégrales est plus performante, car plus rapide (environ un facteur 3) pour une précision équivalente, qu'un calcul purement basé sur des solutions analytiques.

Les méthodes intégrales sont des méthodes à interactions totales, par conséquent elles génèrent des matrices pleines qui sont coûteuses à calculer en temps processeur. Nous proposons au chapitre suivant de paralléliser ces calculs afin de les accélérer. Le point central de cette étude sera l'investigation des apports des cartes graphiques.

7 Références

[Aimé 09] J. Aimé, « Rayonnement des convertisseurs statiques. Application à la variation de vitesse », Thèse de doctorat, INPG, Grenoble, France, 2009.

[Ardon 10a] V. Ardon, J. Aime, O. Chadebec, E. Clavel, J-M. Guichon, E. Vialardi, « EMC Modeling of an Industrial Variable Speed Drive with an Adapted PEEC Method », IEEE transactions on magnetics, vol. 46, no. 8, pp. 2892-2898, 2010.

[Ardon 10b] V. Ardon, « Méthodes numériques et outils logiciels pour la prise en compte des effets capacitifs dans la modélisation CEM de dispositifs d'électronique de puissance », Thèse de doctorat, INPG, Grenoble, France, 2010.

[CEDRAT] CEDRAT, logiciel InCa3D, www.cedrat.com

[Chadebec 01] O. Chadebec, « Modélisation du champ magnétique induit par des tôles, identification de l'aimantation - Application à l'immunisation en boucle fermée d'une coque ferromagnétique », Thèse de doctorat, INPG, Grenoble, France, 2001.

[Cipra 00] B. Cipra, « The Best of the 20th Century : Editors Name Top 10 Algorithms », SIAM News 33(4), 1, 2000.

[Coulomb 12] J-L. Coulomb, « Numerical Design Of Experiments and Optimization », http://forge-mage.g2elab.grenoble-inp.fr/project/got

[Durand 64] E. Durand, « Electrostatique, Tome I - Les distributions », Masson, chap. Distributions superficielles, p. 13-225, 1964.

[Durand 66] E. Durand, « Electrostatique, Tome II – Problèmes généraux conducteurs », Masson, chap. Conducteurs, p. 152-233, 1966.

[Fletcher 76] R. Fletcher, « Conjugate gradient methods for indefinite systems », Lecture Notes in Mathematics, vol. 506, Numerical Analysis, pp. 73-89, 1976.

[Graglia 93] R. Graglia, « On the Numerical Integration of the Linear Shape Functions Times the 3-D Green's Function or its Gradient on a Plane Triangle », IEEE Transactions on Antennas and Propagation, vol. 41, no. 10, Oct. 1993.

[Greengard 87] L. Greengard and V. Rokhlin, « A fast algorithm for particle simulations, Journal of Computational Physics », vol. 73, Issue 2, pp. 325-348, Dec 1987.

[Greengard 99] H. Cheng, L. Greengard, and V. Rokhlin, « A Fast Adaptive Multipole Algorithm in Three Dimensions », Journal of Computational Physics 155, 468–498, 1999.

[Guibert 09] A. Guibert, « Diagnostic de corrosion et prédiction de signature électromagnétique de structures sous-marines sous protection cathodique », Thèse de Doctorat, INGP, Grenoble, France, 2009.

[Hachi-Ashtiani 98] B. Hachi-Ashtiani, « Méthodes d'assemblage rapide et de résolution itérative pour un solveur adaptatif en équations intégrales de frontières destiné à l'électromagnétisme », Thèse de Doctorat, Ecole Centrale Lyon, Lyon, France, 1998.

[Hestenes 52] M. R. Hestenes, E. Stiefel, « Methods of Conjugate Gradients for Solving Linear Systems », *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, Dec. 1952.

[Janssen 10] J.L.G. Janssen, J.J.H. Paulides and E.A. Lomonova, « 3D analytical field calculation using triangular magnet segments applied to a skewed linear permanent magnet actuator », *COMPEL: The International Journal for Computation and Mathematics in Electrical and Electronic Engineering*, vol. 29, no. 4, pp. 984-993, 2010.

[Krahenbuhl 83] L. Krahenbuhl, « La méthode des équations intégrales de frontière pour la résolution des problèmes de potentiel en électrotechnique, et sa formulation axisymétrique », Thèse de Doctorat, Ecole Centrale Lyon, Lyon, France, 1983.

[Le Duc 11] T. Le Duc, « Développement de méthodes intégrales de volume en électromagnétisme basse fréquence. Prise en compte des matériaux magnétiques et des régions minces conductrices dans la méthode PEEC », Thèse de Doctorat, Université de Grenoble, Grenoble, France, 2011.

[Reinauer 11] V. Reinauer, T. Wendland, C. Scheiblich, R. Banucu, « Object-Oriented Development and Runtime Investigation of 3-D electrostatic FEM problems in Pure Java », Proceeding of CEFC 2010 Conference, to be published in IEEE Trans. Mag., 2011.

[Rjasanow 07] S. Rjasanow, O. Steinbach, « The Fast Solution of Boundary Integral Equations », Springer, 2007.

[Ruehli 74] A. Ruehli and A. Cangellaris, « Equivalent Circuit Models for Three-Dimensional Multiconductor Systems », IEEE Transactions on Microwave Theory and Techniques, vol. 22, no. 3, pp. 216-221, Mar. 1974.

[Saad 86] Y. Saad and M.H. Schultz, « GMRES : A generalized minimal residual algorithm for solving nonsymmetric linear systems SIAM », Journal on Scientific and Statistical Computing, vol. 7, no 3, pp. 856-869, 1986.

[Saad 00] Y. Saad, « Iterative Methods for Sparse Linear Systems - Second Edition », SIAM, 2000.

[Scheiblich 09] C. Scheiblich, V. Kolitsas and W. M. Rucker, « Compression of the Radiative Heat Transfer BEM Matrix of an Inductive Heating System Using a Block-Oriented Wavelet Transform, » IEEE Trans. Magn., vol. 47, no. 3, pp. 1712-1715, Mar. 2009.

[Rubeck 11] C. Rubeck, J-P. Yonnet, B. Delinchant, O. Chadebec, « Calcul analytique du potentiel et du champ magnétostatique créés par un aimant permanent de forme polyédrique uniformément aimanté », Electrotechnique du Futur(EF) 2011, France, 2011.

[Weisstein 05] E.W. WEISSTEIN, « Legendre–Gauss quadrature », *MathWorld – A Wolfram Web Resource*, http://mathworld.wolfram.com/Legendre-GaussQuadrature.html, 2005.

[Wilton 79] S. Rao, A. Glisson, D. Wilton, and B. Vidula, « A simple numerical solution procedure for statics problems involving arbitrary-shaped surfaces, » IEEE Trans. Antennas Propagat. vol. 27, no. 5, pp. 604–608, Sep 1979.

Chapitre III

Parallélisation d'une formulation intégrale sur processeurs graphiques

Sommaire

Résumé

Les méthodes intégrales génèrent des matrices pleines qui sont très coûteuses à calculer en temps processeur. Nous proposons dans ce chapitre de réduire les temps de calcul en utilisant plusieurs processeurs, c'est le parallélisme. Les architectures parallèles exploitées sont le PC multicoeur, le cluster de PCs et le GPGPU. Plusieurs stratégies sont confrontées pour le calcul de la matrice d'interaction et sa résolution.

1 Introduction

Les méthodes intégrales sont des méthodes à interaction totale : elles génèrent des matrices de systèmes d'équations pleines. Le calcul de ces matrices est de complexité parabolique, ainsi que la résolution itérative du système d'équations associé. Nous proposons dans ce chapitre d'accélérer les modélisations grâce au parallélisme.

Nous avons au chapitre précédent développé une formulation intégrale en potentiel associé à une méthode de collocation à l'ordre 0. L'assemblage est ici très simple, chaque interaction génère un terme de la matrice d'interaction. Ces termes sont donc indépendants les uns des autres, ils peuvent alors être calculés simultanément : le parallélisme est alors optimal. Nous utilisons plusieurs architectures parallèles, chaque architecture nous conduit à développer une stratégie de parallélisation adaptée, notamment sur les points clefs tels que la gestion de la mémoire et la synchronisation des tâches.

Nous commençons naturellement par paralléliser les codes sur CPU multicoeur car c'est l'architecture la plus courante. Nous construisons ensuite un petit cluster de PCs en réseau. L'intérêt est de pouvoir cumuler les ressources processeurs et mémoires de plusieurs machines. Le point central de ce chapitre est ensuite la parallélisation sur l'architecture GPGPU. Nous testons différentes stratégies pour accélérer le calcul de la matrice d'interaction ainsi que la résolution itérative.

2 Présentation des cas tests

2.1 Problème physique

Le problème à résoudre est le calcul de la capacité propre de la sphère maillée en triangles présentée au chapitre II (5.1). Nous rappelons qu'il existe une solution analytique exacte ce qui est intéressant pour estimer les erreurs de nos modélisations.

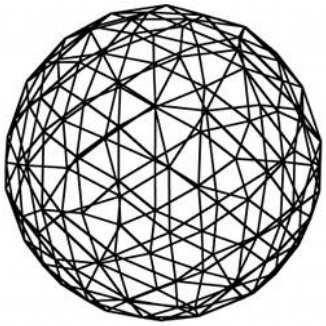


Figure 1 : Sphère maillée en triangles.

Nous utilisons plusieurs maillages de sphères pour nos tests. Nous présentons les différents maillages et la dénomination du cas correspondant dans le tableau présenté à la Figure 2.

Nombre d'éléments du maillage	Dénomination du cas test
9068	Cas9000
20602	Cas20000
29650	Cas30000

Figure 2 : Dénomination des cas tests en fonction des maillages.

2.2 Architectures parallèles testées

Nous présentons sur la Figure 3 les différentes architectures parallèles que nous allons tester pour accélérer le calcul de la matrice d'interaction et sa résolution.

Les architectures parallèles présentées sur la Figure 3 A) et B) sont un PC multicoeur respectivement à mémoire partagée et à mémoire distribuée. Une programmation parallèle à mémoire partagée signifie que toutes les tâches peuvent accéder aux mêmes plages mémoires. C'est la programmation la plus simple à mettre en œuvre, nous utilisons le multi-threading natif de Java. Dans une programmation parallèle à mémoire distribuée, chaque tâche est un processus indépendant des autres. Les mémoires ne sont pas visibles par les autres tâches et les communications se font via un protocole réseau (le réseau est ici interne au PC, les communications sont plus rapides qu'à travers une carte réseau, mais plus lentes qu'un accès direct à une mémoire partagée).

Nous testerons ensuite un petit cluster de PCs (Figure 3, C). Il s'agit également d'une programmation à mémoire distribuée. Les communications sont effectuées à travers le réseau local du laboratoire.

Nous finirons avec la parallélisation sur architecture GPGPU (Figure 3, D). Ici la carte graphique est pilotée par un ordinateur hôte à travers le BUS. Cette architecture est également à mémoire distribuée.

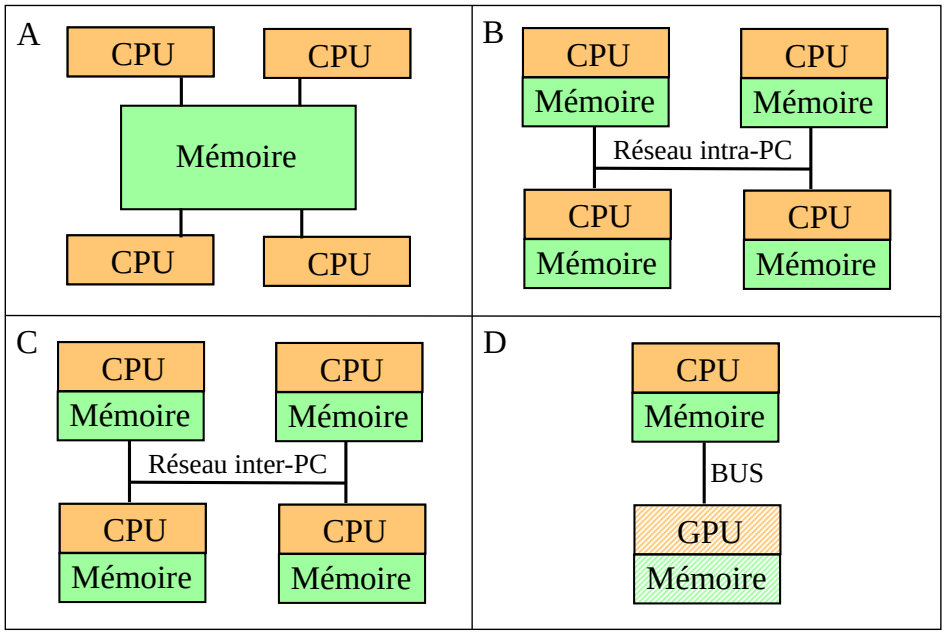


Figure 3 : Architectures parallèles testées. A) PC multicoeur à mémoire partagée, B) PC multicoeur à mémoire distribuée, C) Cluster de PCs, D) GPGPU.

2.3 Configurations matérielles

L'ordinateur principal est un PC sous Ubuntu GNU/Linux en 64 bits. Il possède 4 Go de RAM et il est équipé d'un processeur Intel Xeon cadencé à 2,67 GHz, ce dernier possède 4 cœurs de calcul. Par défaut nos calculs sur CPU se feront en monocoeur et en double précision.

Le réseau testé dispose d'une vingtaine d'ordinateurs partagés. Nous choisissons les 6 ordinateurs les plus récents pour réaliser l'expérience. Chaque machine dispose de 2 Go de mémoire allouée à la machine virtuelle Java. Les processeurs sont des Core2Duo cadencés à 3,00GHz, des Core i7 à 3,4GHz et même des Core2Quad à 3,00GHz.

La carte graphique par défaut est une carte de calcul Nvidia Tesla C1060, elle possède 4Go de RAM et 240 cœurs de calcul cadencés à 1,30 GHz répartis sur 30 multiprocesseurs de 8 cœurs chacun. Cette carte permet des calculs en simple et en double précision. Nous effectuerons également quelques tests sur une carte graphique d'entrée de gamme, à savoir une Nvidia Geforce 320M installée dans un MacBook. Elle possède 250 Mo de mémoire partagée (c'est-à-dire empruntée à la mémoire globale du MacBook), 48 cœurs de calcul à 0,95 GHz, soit 8 multiprocesseurs de 8 cœurs chacun. Notons que cette carte graphique est dédiée à l'affichage et par conséquent toutes ses ressources ne sont pas disponibles pour effectuer des calculs scientifiques. De plus elle ne permet que des calculs en simple précision.

2.4 Performances de référence

Nous mesurons deux temps de calcul lors de nos expériences : le temps d'intégration qui comprend les temps de génération des tables (points de Gauss et de collocation, jacobiens, poids de Gauss) et du temps de calcul de la matrice d'intégration, le deuxième temps et le temps de résolution du système d'équations linéaires. Par défaut, la matrice d'interaction est construite via des intégrations numériques à 7 points de Gauss par élément. La diagonale est remplacée par des solutions analytiques. La résolution est effectuée par un solveur GMRES, le critère de convergence est de 1e-9.

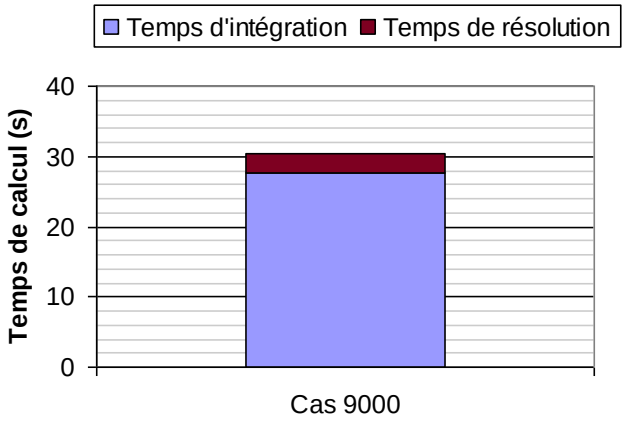


Figure 4 : Temps d'intégration et de résolution du Cas9000.

Nous présentons à la Figure 4 les temps d'intégration et de résolution du Cas9000. Les paramètres expérimentaux sont ceux par défaut. Nous remarquons que le temps

d'intégration est supérieur d'un ordre de grandeur au temps de résolution, c'est donc ce temps qu'il faut réduire en priorité. Nous ne présentons pas les Cas20000 et Cas30000 car ils ne sont pas réalisables en double précision avec seulement 4Go de mémoire vive.

3 Parallèlisation sur PC multicoeur

Le parallélisme le plus répandu est évidement l'utilisation de plusieurs processeurs au sein d'un même ordinateur [Buchau 08]. Nous proposons d'étudier deux types de programmation parallèle : à mémoire partagée et à mémoire distribuée. Le cas test est le Cas9000.

3.1 Architecture parallèle à mémoire partagée

Le partitionnement adopté ici consiste à calculer plusieurs lignes de la matrice d'interaction simultanément. De même pour la résolution nous calculons plusieurs termes du résidu simultanément lors des produits matrice-vecteur.

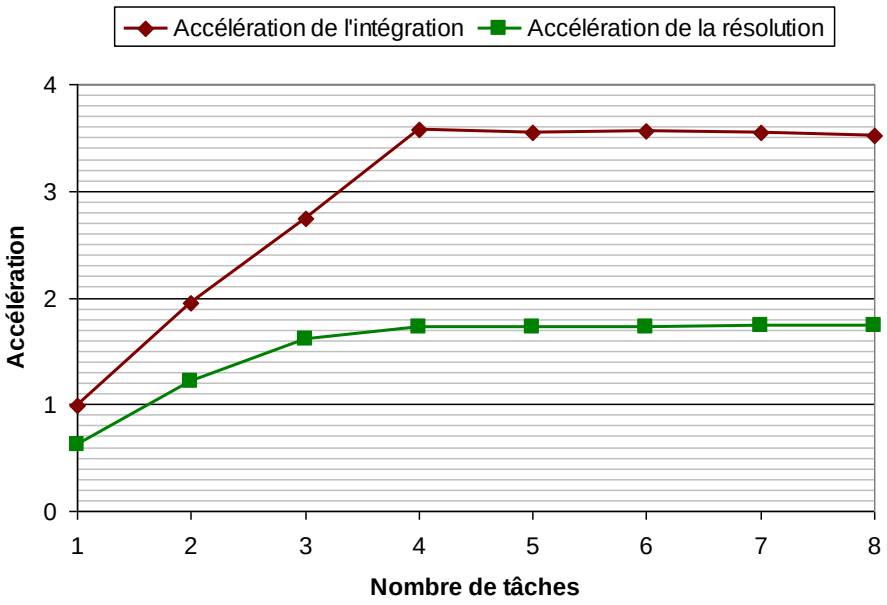


Figure 5 : Accélération des temps d'intégration et de résolution en fonction du nombre de tâches sur une architecture parallèle à mémoire partagée.

Nous présentons sur la Figure 5 l'accélération[1] des temps de calcul d'une matrice d'interaction et de sa résolution en fonction du nombre de tâches définies, la référence étant l'utilisation d'une seule tâche.

L'accélération de l'intégration possède deux régimes. Le premier croit quasi linéairement jusqu'au nombre de cœurs de calcul (ici 4). L'allure de la courbe se situe bien entre les lois de Hamdhal [Amdahl 67] et de Gustafson [Gustafson 88] avec une nette tendance pour le comportement linéaire décrit par la loi de Gustafson. Le deuxième régime est constant, légèrement décroissant, définir d'avantage de tâches qu'il n'y a de cœurs de calcul pourrait permettre d'effectuer plus de calcul (cf. hyperthreading), mais ce n'est pas le cas ici et cette stratégie coûte plus cher sur la gestion des tâches d'où la baisse des performances.

La parallélisation de la résolution est beaucoup moins performante. Nous n'obtenons même pas une accélération de 2. L'explication est que la résolution contient une part importante de séquentialité, en effet seul le calcul du résidu est parallélisé, tout l'algorithme GMRES demeure séquentiel.

3.2 Architecture parallèle à mémoire distribuée

L'algorithme correspondant est présenté en détail au point suivant. La matrice d'intégration est ici divisée en sous matrices qui sont distribuées aux différentes tâches. Pour la résolution itérative, chaque tâche effectuera les calculs des résidus partiels associés à toutes les sous-matrices qui lui ont été attribuées.

La Figure 6 montre l'accélération des temps d'intégration et de résolution en fonction du nombre de tâche. Le cas test est toujours de Cas9000. Comme précédemment nous voyons un régime croissant jusqu'au nombre de cœur du CPU suivi d'un régime décroissant. Le régime croissant n'est pas linéaire ici, l'accélération est dominée par la loi de Hamdhal : les temps de communication réseau nuisent à la parallélisation. Les transferts de données à travers un réseau sont de plusieurs ordres de grandeur plus lents que ceux passant à travers un bus. L'accélération maximale dépasse à peine 2 pour 4 cœurs de

[1] Rappel : l'accélération est le rapport des temps de calcul entre le programme séquentiel et sa version parallèle.

calcul, ce n'est pas très intéressant. Le deuxième régime est décroissant car augmenter le nombre de tâche augmente également les communications réseaux et donc diminue l'efficacité de la parallélisation.

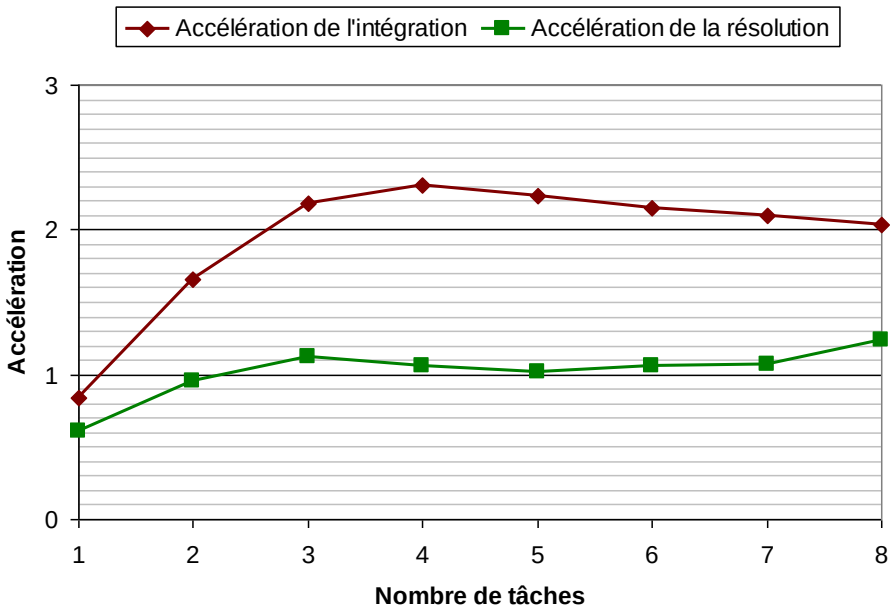


Figure 6 : Accélération des temps d'intégration et de résolution en fonction du nombre de tâches sur une architecture parallèle à mémoire distribuée.

La résolution n'est pas vraiment accélérée, l'explication vient de la synchronisation des tâches. Chaque tâche effectue ses calculs de résidus partiels. Elles sont ensuite synchronisées pour calculer le résidu total. Par conséquent le temps de calcul du résidu est donné par la tâche la plus lente, additionné au temps de la synchronisation des tâches.

3.3 Conclusion

La programmation parallèle par mémoire partagée est plus performante que celle par mémoire distribuée. Nous nous attendions à ce résultat car dans le deuxième cas la gestion des connexions réseaux (même si elles restent locales au PC) induit naturellement un coût supplémentaire. Ce coût est même relativement élevé. La scalabilité[2] est ici de 3,5 pour 4 cœurs soit 86% pour la programmation à mémoire partagée, et elle est de 58% pour la

[2] Rappel : la scalabilité est le rapport de proportionnalité entre l'accélération et le nombre de cœurs de calcul. Le cas idéal est 100%, c'est-à-dire une parallélisation parfaite.

programmation à mémoire distribuée. De plus la programmation parallèle à mémoire partagée est plus simple à mettre en œuvre, en effet la plupart des langages de programmation gèrent le multitâche.

Une dernière remarque, dorénavant nous définirons autant de tâches qu'il y a de cœurs de calcul. Essayer de surexploiter le pipelining en définissant plus de tâches qu'il n'y a de cœurs de calcul n'est pas efficace, cette approche s'est même montrée moins performante.

4 Cluster de PCs

Nous avons utilisés plusieurs cœurs de calcul au sein d'un même ordinateur. Cependant le nombre de cœurs reste limité, sans oublier la quantité de mémoire disponible. Une solution pour obtenir plus de cœurs de calcul et de mémoire est d'utiliser plusieurs ordinateurs reliés via un réseau [Buchau 08]. Le cumul de toutes les ressources informatiques permet de traiter des problèmes plus importants (le facteur limitant étant très souvent la quantité de mémoire).

4.1 Architecture du cluster

L'architecture réseau retenue pour cette expérience est le classique duo serveur-client issu du protocole tcp/ip accessible via la programmation par sockets. Un socket est un objet qui gère l'ouverture des ports, les transmissions, etc. Il est disponible sur la plupart des langages de programmation. L'intérêt de cette structure est que le développeur évite de programmer directement la couche réseau du système d'exploitation.

Le modèle de fonctionnement du programme de calcul de capacité est présenté à la Figure 7. Le chef d'orchestre est le serveur : il s'occupe de la gestion des connexions réseaux, du pré et du post traitement du problème. Tous les calculs sont effectués par les clients. La première étape est l'attente de la connexion des clients. Une fois les clients connectés, ces derniers lui communiquent les ressources qu'ils possèdent, ainsi les tâches seront distribuées en fonction des capacités de chaque client (notamment la quantité de mémoire). Dans un deuxième temps, la matrice d'interaction est partitionnée. Les sous matrices créées sont ensuite distribuées sur les différents clients afin d'y être calculées. Ces sous matrices une fois calculées restent dans la mémoire des clients. Pour la phase de

résolution itérative, le vecteur solution est transmis à tous les clients. Ces derniers calculent alors leur résidu partiel respectif en fonction des éléments de la matrice dont ils disposent. Ces résidus partiels sont ensuite transmis au serveur qui les additionne pour obtenir le résidu total. Une fois que la solution a convergé, les connexions vers les clients sont fermées et ces derniers sont arrêtés. Le post traitement est assuré par le serveur.

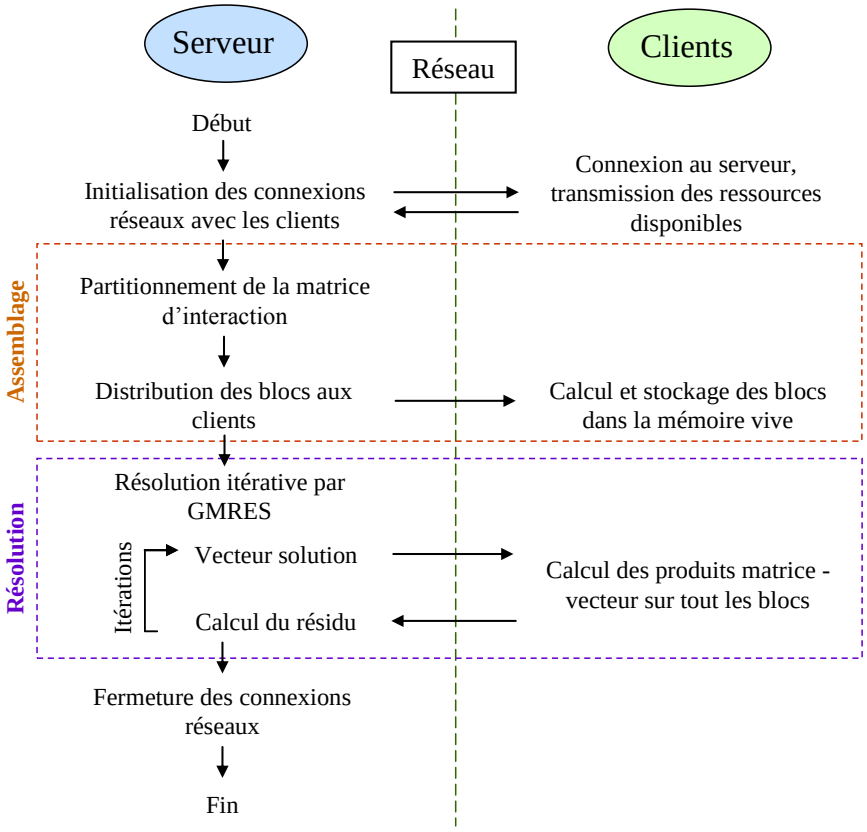


Figure 7 : Interconnexions entre le serveur et les clients à travers un réseau.

4.2 Accélération à nombre de degrés de liberté constant

L'expérience est effectuée à nombre de degrés de liberté constant, ici le Cas9000. La matrice d'interaction est partitionnée en 100 blocs de taille 900x900 éléments matriciels environ (choix arbitraire) qui sont distribués sur les ordinateurs en réseau. Autant de processus clients sont lancés qu'il y a de cœurs de calcul sur chaque machine, la prise en compte du multicoeur est ici très simple.

La Figure 8 présente les accélérations des calculs d'intégration et de résolution en fonction du nombre de tâches réseaux. Contrairement au multi-CPU, l'accélération de l'intégration connaît un maximum d'à peine 2 à 12 processeurs puis décline. Le temps de

la gestion d'un réseau nuit clairement à la parallélisation. Le réseau utilisé n'est pas adapté à cet usage.

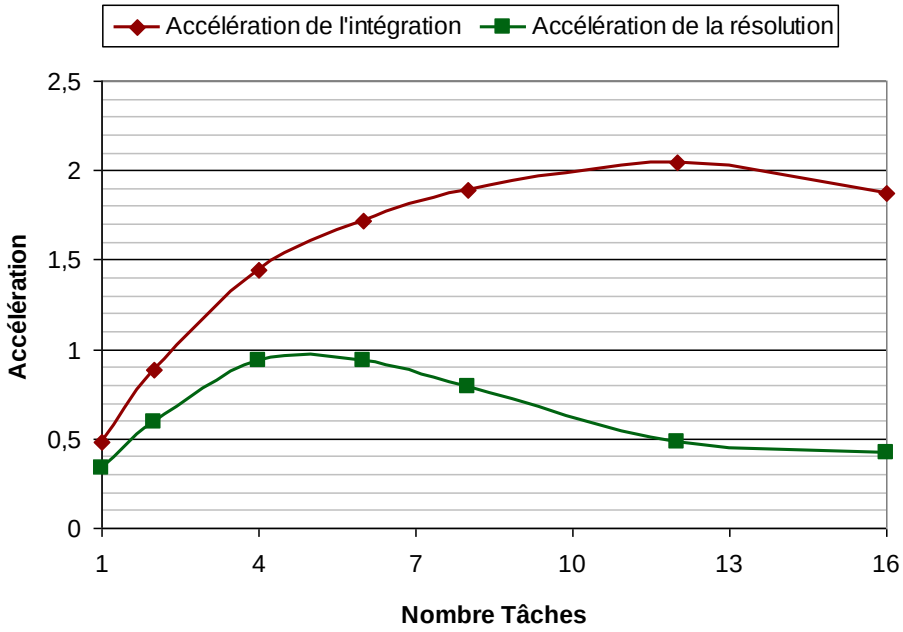


Figure 8 : Accélération des calcul d'intégration et de résolution en fonction du nombre de processeurs disponibles sur des ordinateurs relié en réseau.

Tentons de modéliser l'accélération de l'intégration. Nous pouvons définir le temps de calcul total comme étant la somme des temps des communications réseaux et des temps de calcul :

$$T_{total} = n.T_{réseau} + \frac{T_{calcul}}{n} \quad (1)$$

Avec n le nombre de clients. Nous faisons l'hypothèse d'une proportionnalité directe entre les temps de communications et le nombre de connections réseaux. Nous considérons également le calcul comme étant parfaitement parallélisable, le temps de calcul est donc réduit d'un facteur n. Nous pouvons alors écrire l'accélération :

$$Acc = \frac{T_{calcul}}{n.T_{réseau} + \frac{T_{calcul}}{n}} \quad (2)$$

Nous présentons l'allure de la courbe du modèle à la Figure 9. Nous définissons arbitrairement le temps réseau à 1 et le temps de calcul à 100. Nous retrouvons bien

l'allure de la courbe expérimentale. Par conséquent les gains que peut apporter un petit cluster sont limités par le coût de la gestion du réseau.

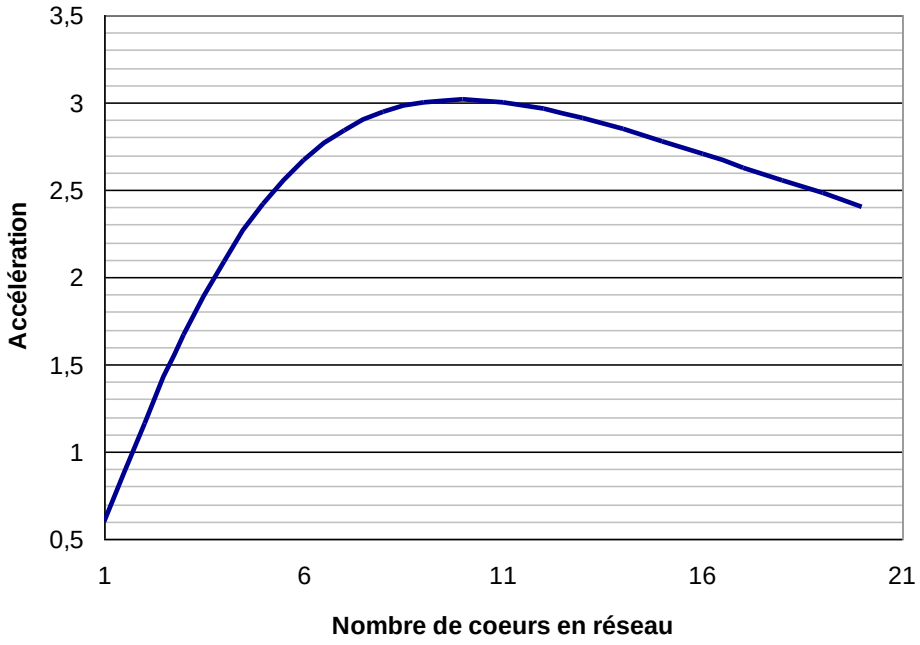


Figure 9 : Allure de la courbe modélisant l'accélération du calcul de la matrice d'interaction via un réseau. Ici Treseau=1, Tcalcul=100 (arbitraire).

La courbe de résolution illustre l'importance du coût des communications. L'accélération cesse d'augmenter à partir de 6 processeurs, soit ici 3 ordinateurs reliés en réseau, et n'atteint même pas 1. Les calculs des résidus devant être effectués sur tous les ordinateurs puis synchronisés sur le serveur, le temps de calcul est finalement fixé par l'ordinateur le plus lent ainsi que par la congestion du réseau. C'est pourquoi l'accélération de la résolution est si peu intéressante.

4.3 Augmentation du nombre de degrés de liberté

La deuxième expérience que nous proposons consiste à utiliser un nombre constant d'ordinateurs (ici 5, soit un total de 12 cœurs de calcul) et d'étudier les temps de calcul en fonction de la monté en degrés de liberté. L'idée est de cumuler les ressources de calcul (notamment mémoire) des machines afin d'effectuer des calculs qu'il serait impossible de réaliser sur une seule machine.

Nous présentons à la Figure 10 les temps d'intégration et de résolution en fonction du nombre d'éléments. Nous affichons également les courbes de références obtenues en monocoeur. Les courbes sont toutes paraboliques, les gains en vitesse d'intégration entre le monocoeur et le cluster se situent autour de 3 pour 13.000 éléments. C'est peu sachant que le cluster possède 12 cœurs de calcul. Par contre le cluster permet de traiter des problèmes à nombre d'éléments beaucoup plus élevé.

Notons que la courbe de résolution réseau possède une composante linéaire très importante due au coût des communications. Nous pouvons estimer que le temps d'une itération GMRES est la somme des temps de transferts des vecteurs $O(N)$ sur le réseau et des temps du produit matrice vecteur $O(N^2)$.

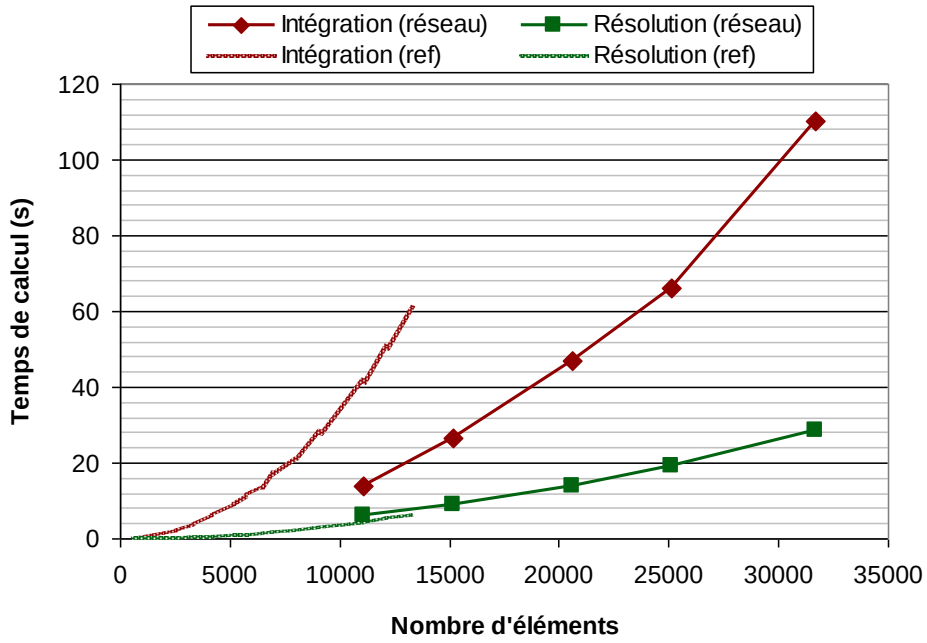


Figure 10 : Temps d'intégration et de résolution en fonction du nombre d'éléments (5 PCs en réseaux). Les courbes de références sont obtenues en monocœur.

4.4 Conclusion

L'utilité d'un petit cluster de calcul est bien illustrée ici car il n'est pas possible de traiter des problèmes supérieurs à 10.000 degrés de liberté sur une seule machine, les ressources mémoires étant insuffisantes. Concernant les ressources processeurs, avoir un grand nombre de cœurs de calcul via un réseau n'est pas forcement intéressant à cause de la partie réseau qui limite fortement les gains. L'idéal pour fortement accélérer les calculs

serait d'avoir un grand nombre de processeurs sur une architecture à mémoire partagée, voici pourquoi nous allons investiguer l'utilisation des cartes graphiques.

5 Calcul de la matrice d'interaction sur processeurs graphiques

5.1 Approche pseudo massivement parallèle

Une première approche naïve consiste à remplacer la bibliothèque de calcul matricielle Java par un équivalent qui exploite le GPU. Ces opérations de haut niveau sur des vecteurs de plusieurs milliers d'éléments devraient être fortement accélérées par les centaines de cœurs de calcul du GPU. Avec N le nombre d'éléments, il y a donc N lignes de la matrice d'interaction à calculer vectoriellement. Cette approche très simple (une bibliothèque matricielle en remplace une autre) ne s'est malheureusement pas montrée efficace. Pire, cette approche GPU est d'un ordre de grandeur plus lente que le calcul sur CPU...

Malgré le fait qu'une opération matricielle ou vectorielle donnée est plus rapide sur GPU que sur CPU, il existe des temps de latence entre chaque opération sur GPU tels que les temps de construction des grilles de calcul, des chargements des noyaux CUDA, et des transferts de données entre le GPU et l'hôte. Le GPU effectue finalement peu de calcul entre toutes ces opérations de communication qui sont bien trop séquentielles. Il est donc nécessaire de réduire le nombre de ces opérations. L'idéal serait de n'avoir qu'une phase de transferts de données (maillage, points de Gauss, etc.) et qu'une seule instruction pour la construction de la matrice d'interaction. La solution que nous allons appliquer est d'écrire un noyau CUDA (programmation bas niveau) dédié à notre formulation intégrale.

5.2 Noyau CUDA dédié au calcul d'intégrales

La stratégie retenue pour occuper le calculateur graphique au maximum est d'effectuer tous les calculs d'intégrales simultanément. La grille de calcul correspondante est présentée sur la Figure 11. Chaque thread effectue un calcul intégral de la matrice [Lezar 10]. La grille est en deux dimensions, ainsi les coordonnées globales des threads dans la grille de calcul correspondent aux coordonnées des éléments de la matrice d'interaction.

Les deux premières boucles imbriquées de l'algorithme classique (parcours des lignes et des colonnes) sont donc remplacées par la topologie de la grille de calcul. Il ne reste alors dans chaque thread uniquement le calcul de l'intégrale, c'est-à-dire la boucle nécessaire à l'intégration numérique par points de Gauss. Nous avons également testé une version avec une grille en 3D, la troisième boucle est éclatée et chaque thread effectue alors une partie de l'intégration de Gauss. Cette approche ne s'est pas montrée satisfaisante à cause de l'opération de réduction requise pour la sommation des termes de l'intégrale.

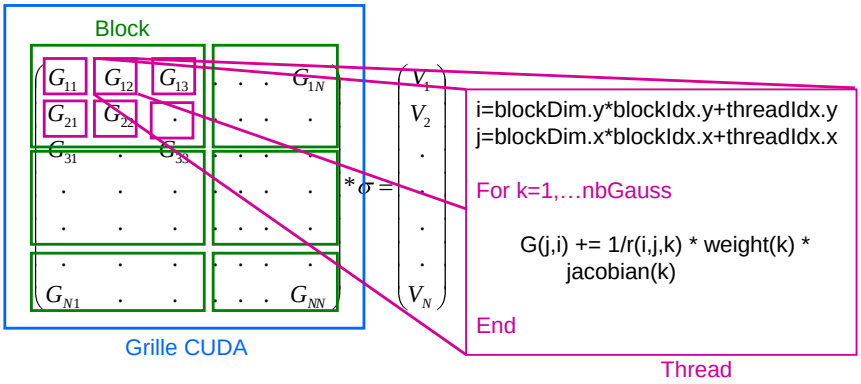


Figure 11 : Grille de calcul CUDA pour le calcul de la matrice d'interaction. Chaque thread effectue un calcul intégral.

Tout comme dans le cas de l'implémentation Java, les tables des points de Gauss, des poids, des déterminants des jacobiens et des points de collocation sont calculées au préalable. Chacune de ces opérations est effectuée par un noyau CUDA dédié. Les grilles de calcul associées sont toutes en une dimension, chaque thread calcule les points de Gauss, les poids, les jacobiens et le point de collocation associé à un seul élément du maillage. Une fois ces tables calculées, le noyau CUDA de calcul d'intégrales est exécuté. Un noyau CUDA pour le calcul analytique de la diagonale a été développé. Nous avons également développé un noyau de calcul analytique de l'ensemble de la matrice d'interaction. Les noyaux CUDA ont été développés pour des calculs en simple et en double précision.

Les transferts de données sont donc fortement réduits dans cette approche de calcul sur GPU. Seul le maillage est transféré de l'hôte vers le GPU. Les tables intermédiaires et la matrice d'interaction sont construites directement dans la mémoire graphique.

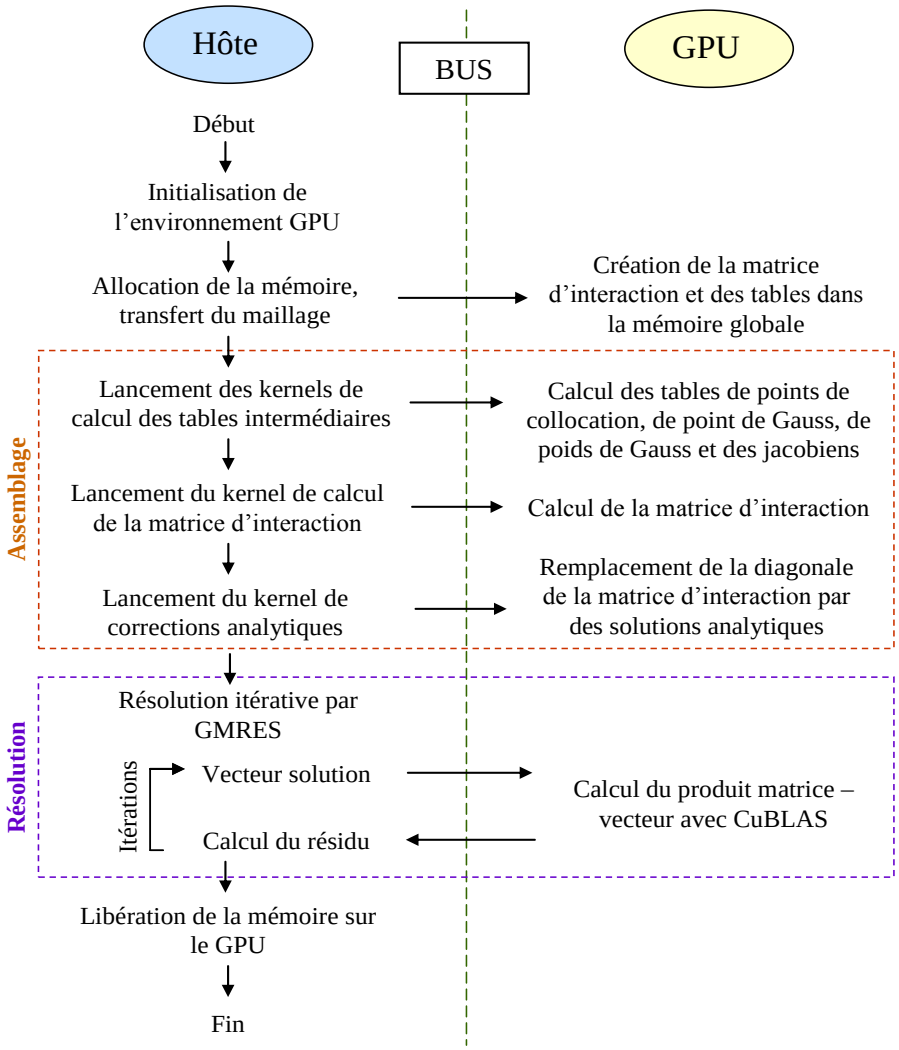


Figure 12 : Interconnexions entre l'ordinateur hôte et le GPU.

5.3 Performances

La Figure 13 montre les temps de construction de la matrice d'interaction en fonction du nombre d'éléments. Nous comparons les performances des architectures CPU (toujours en monocoeur) et GPU, ainsi que les méthodes d'intégration, ici analytique et numérique à 7 points de Gauss avec diagonale analytique. La complexité parabolique du calcul de la matrice est bien illustrée. Comme précédemment, nous retrouvons que le calcul entièrement analytique est plus coûteux que le calcul numérique par points de Gauss, que ce soit sur CPU ou GPU.

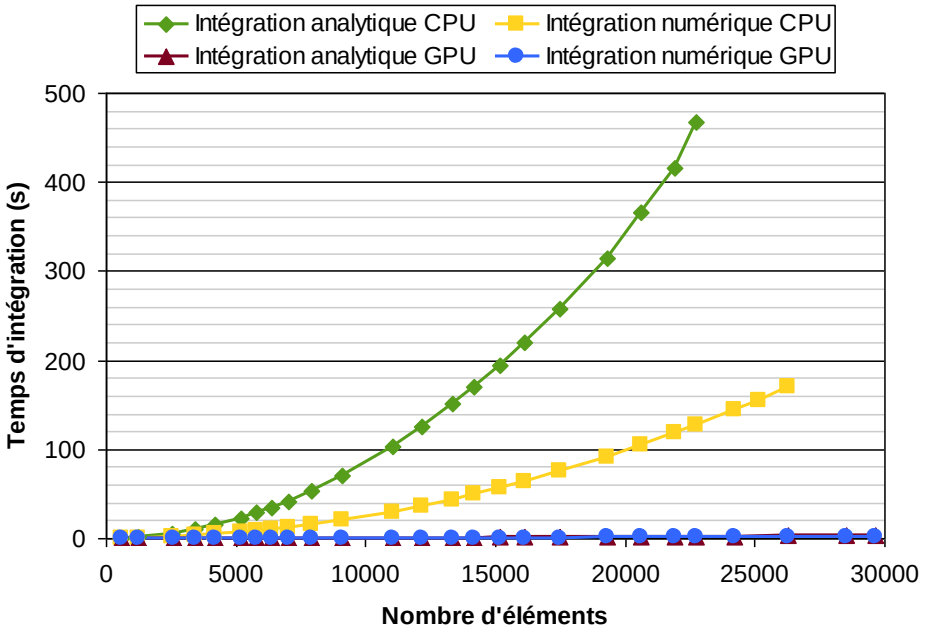


Figure 13 : Temps de construction de la matrice d'interaction en fonction du nombre d'éléments pour différentes méthodes d'intégrations et d'architectures : CPU monocoeur et GPU, intégration analytique et numérique à 7 points de Gauss corrigée, calcul en simple précision.

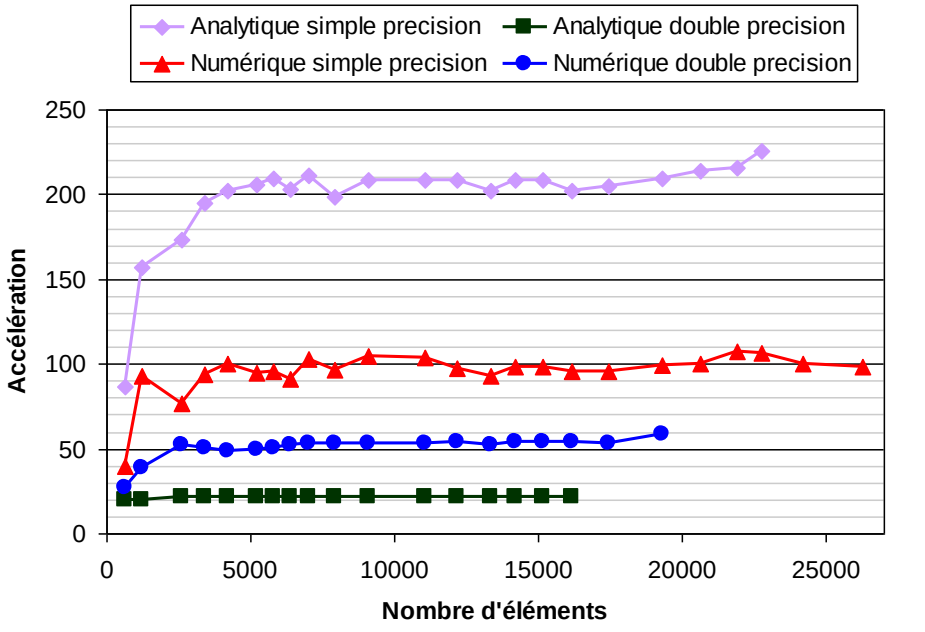


Figure 14 : Accélération du calcul de la matrice d'interaction en fonction du nombre d'éléments entre architectures CPU monocoeur et GPU. Les calculs numériques et analytiques, simple et double précision sont comparés.

Nous présentons à la Figure 14 les accélérations des calculs de matrices d'interaction entre CPU et GPU. Nous comparons les méthodes d'intégration et les précisions de calcul. Le calcul le plus accéléré par l'utilisation des cartes graphiques est le calcul analytique en simple précision, il est ici 200 fois plus rapide que sur CPU. Ce résultat est à relativiser, en effet le calcul analytique sur CPU est très peu performant car non vectorisé. L'ordre de grandeur de l'accélération rencontré dans la littérature est d'environ 50 fois [Lezar 10, D'Ambrosio 11]. Le résultat le plus intéressant est l'accélération d'un facteur 100 pour le calcul numérique en simple précision. Nous pouvons également observer une montée en charge de la puissance de la carte graphique jusqu'à 4000 éléments. En dessous, le calcul n'est pas assez important pour exploiter toute la puissance du GPU. Ces chiffres d'accélération sont à nuancer, en effet nous comparons une programmation objet générique en Java à une notre implémentation bas niveau en CUDA.

5.4 Analyse des temps de calcul GPU

Nous tentons ici de mieux comprendre comment se déroule le calcul sur le GPU afin de proposer des optimisations. Les cartes graphiques de générations GT200 sont connues pour ne pas proposer de calcul en double précision performant. Nous comparons le rapport de vitesse entre les calculs en simples et doubles précisions sur GPU à la Figure 15. L'intégration analytique est 11 fois plus rapide en simple précision qu'en double. L'intégration numérique est quant à elle 3 fois plus rapide en simple précision qu'en double. Ces résultats surprenants de prime abord peuvent nous en apprendre beaucoup sur la manière dont les calculs sont effectués.

Nous présentons sur la Figure 16 une synthèse des principaux facteurs pouvant expliquer les différences de performances entre les calculs en simple et double précision. Nous pouvons estimer que le temps de calcul total est la somme du temps des accès mémoire et du temps de calcul processeur. En double précision, les variables étant deux fois plus larges, le temps des transferts est donc doublé. Chaque multiprocesseur de 8 cœurs dispose uniquement d'une unité de traitement 64 bits capable de traiter des calculs en double précision. Le temps de calcul processeur est donc multiplié par 8 lors du passage en double précision. En ce qui concerne le calcul analytique, ce dernier étant plus compliqué et nécessitant un grand nombre de variables, le passage en double précision a

réduit le taux d'occupation des processeurs de moitié (information fournie par le calculateur d'occupation de Nvidia).

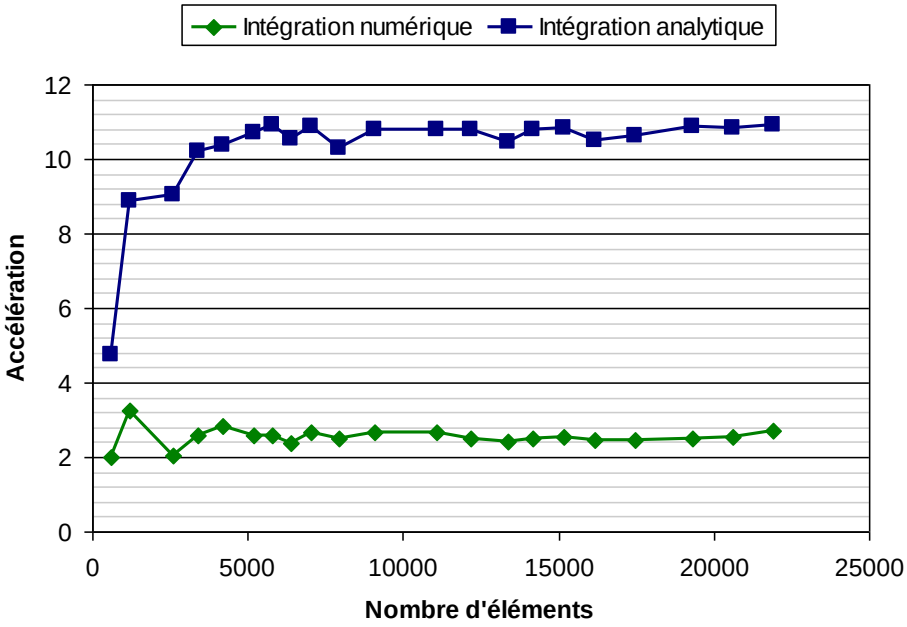


Figure 15 : Accélération entre calculs en simple et double précision sur GPU en fonction du nombre d'éléments dans les cas de l'intégration numérique et analytique.

Par conséquent, le calcul numérique est en théorie de 2 à 8 fois plus lent en double précision qu'en simple, nous obtenons un rapport de 3. Le calcul analytique est en théorie plus lent d'un facteur compris entre 2 et 16, il est ici de 11. Nos résultats sont bien dans les intervalles théoriques, ce qui valide notre analyse. Nous pouvons alors en déduire que le temps de calcul du kernel d'intégration numérique est essentiellement consacré aux accès aux tables intermédiaires. Le calcul des intégrales ne faisant appel qu'aux opérateurs arithmétiques de base (addition, soustraction, multiplication, division). L'intégration analytique est quant à elle beaucoup plus coûteuse en temps processeur. Ce n'est pas étonnant compte tenu de la complexité du calcul, par exemple des logarithmes et des arctangentes doivent être calculés.

En conclusion, pour accélérer le calcul numérique par points de Gauss, il est nécessaire d'optimiser les accès mémoires car ce sont les opérations les plus coûteuses en temps. Le calcul analytique est quant à lui très coûteux en temps processeur, une optimisation des accès mémoire aura peu d'influence sur le temps d'exécution global.

	Dimension des variables	Nombre de cœurs actifs par multiprocesseur	Taux d'occupation des multiprocesseurs
Simple précision	**32 bits**	**8**	**100%**
Double précision	**64 bits**	**1**	**50%**
Ralentissement	**x2**	**x8**	**x2**
Bilan	**Transferts 2 fois plus lents**	**Calculs 16 fois plus lents**	

Figure 16 : Estimation des facteurs expliquant les différences de performances entre les calculs en simple et double précision du noyau de calcul analytique de la matrice d'interaction.

5.5 Optimisations avec la mémoire partagée

Nous venons de voir que le calcul numérique par points de Gauss de la matrice d'interaction se résume surtout à des transferts de données, les optimiser augmenterait les performances. Il existe une mémoire au plus proche des cœurs de calcul appelée mémoire partagée. L'optimisation que nous proposons consiste à nous en servir comme tampon entre la mémoire globale où sont stockées les tables intermédiaires et les registres associés aux cœurs de calcul (Figure 17). Les points de Gauss nécessaires au calcul des coefficients d'interactions d'une même colonne de la matrice sont identiques (idem pour les poids de Gauss et les jacobiens), ainsi que les points de collocation sur une même ligne. Ainsi un block de dimension 16x16 n'a plus besoin que de 16+16=32 accès aux tables au lieu des 16x16=256 sans l'utilisation de la mémoire partagée. Cependant des synchronisations seront requises ce qui pourrait s'avérer plus nuisible que la réduction du nombre d'accès à la mémoire globale.

Nous présentons sur la Figure 18 les temps de construction de la matrice d'interaction par la méthode numérique des points de Gauss. Nous comparons les performances du kernel basique avec celui qui exploite la mémoire partagée. Les performances sont clairement améliorées, le kernel optimisé est 3 fois plus rapide dans le Cas30000. Cet exemple d'optimisation illustre bien la nécessité de bien comprendre les interactions entre l'algorithme et le matériel.

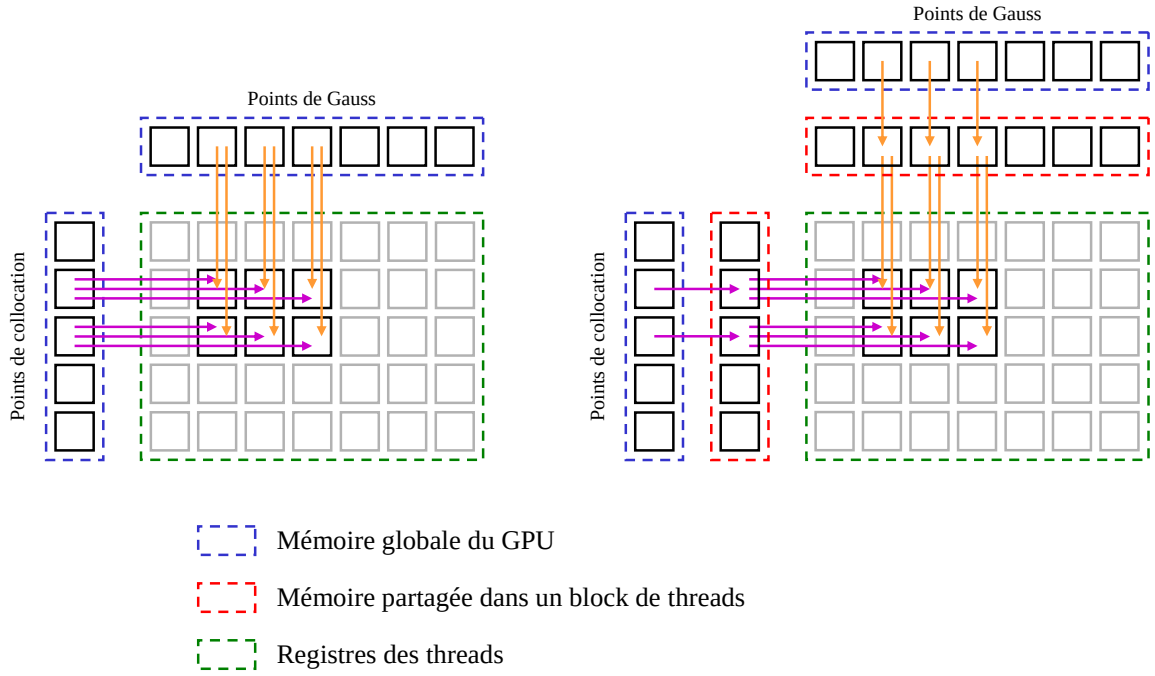


Figure 17 : Optimisation des accès à la mémoire globale du kernel de calcul de la matrice d'interaction. A gauche le kernel d'origine qui exige un grand nombre d'accès à la mémoire globale, à droite le kernel qui utilise de la mémoire partagée pour réduire fortement les accès à la mémoire globale (les accès entre les registres et la mémoire partagée ne sont pas coûteux).

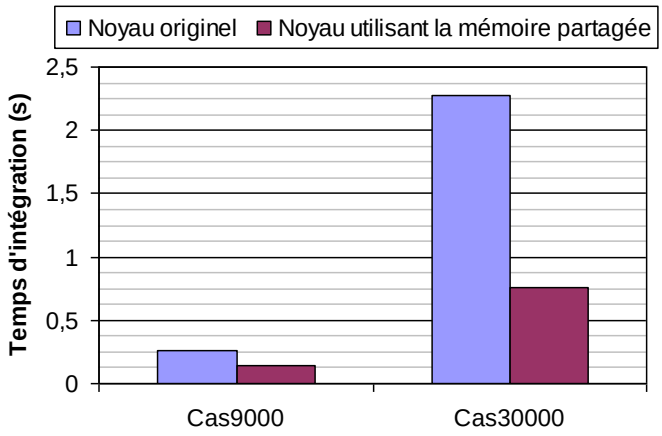


Figure 18 : Temps de construction de la matrice d'interaction par intégration numérique en fonction du nombre d'éléments pour deux versions du noyau CUDA utilisant ou pas de mémoire partagée pour réduire le nombre d'accès mémoire.

5.6 Influence de la topologie de la grille de calcul

La grille de calcul choisie associe un thread à chaque calcul d'intégrale. Ces threads sont regroupés dans des blocks, la particularité des threads d'un même block est qu'ils

partagent une mémoire commune appelée mémoire partagée. La taille des blocks est fixée par le programmeur, c'est une étape importante lors de la définition de la grille de calcul CUDA. Nous proposons ici d'étudier l'influence de la taille des blocks sur le temps de construction de la matrice d'interaction.

Nous définissons des blocks carrés de côtés de taille *n*. La Figure 19 présente les temps d'intégration en fonction de *n* pour le Cas30000, ainsi que l'occupation du GPU en fonction de *n*. Nous observons sur la courbe de gauche que plus les blocks sont petits et plus lente est l'intégration. L'explication vient du diagramme de droite, les petits blocks n'occupent pas suffisamment les cœurs de calcul.

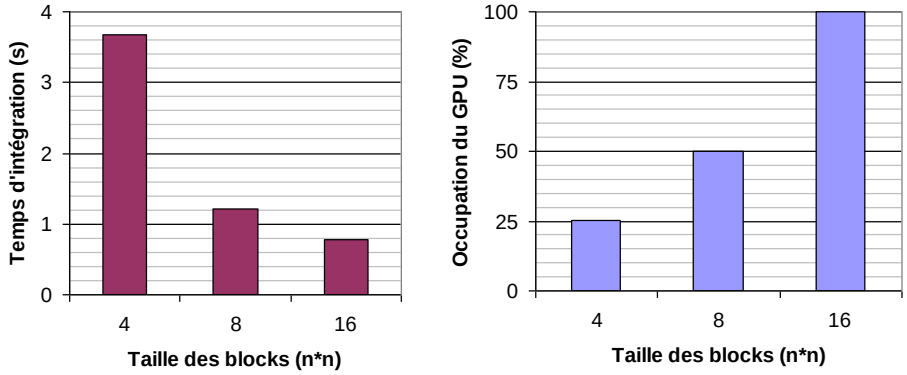


Figure 19 : Influence de la taille des blocks sur le taux d'occupation du GPU et par conséquent les temps d'intégrations de la matrice d'interaction. Cas30000.

En conclusion, il est nécessaire de définir les plus grands blocks possibles afin d'occuper au maximum le GPU. C'est la stratégie que Nvidia recommande [Nvidia 11].

5.7 Stratégie de calcul en fonction de la carte graphique

Une version du programme a été développée pour supporter les cartes graphiques d'entrée et de moyenne gamme, c'est-à-dire des cartes graphiques possédant peu de mémoire vive (Figure 20). Dans ce cas, la matrice d'interaction n'est pas stockable sur la carte graphique. Cette dernière est alors calculée par morceaux sur le GPU et transférée progressivement dans la mémoire de l'hôte. La résolution se fera donc uniquement sur l'ordinateur hôte. Le programme est capable d'interroger la carte graphique pour connaître la quantité de mémoire vive disponible et adapter sa stratégie en fonction de celle-ci. Si la

quantité de mémoire vive est suffisante, alors le système d'équations sera construit et résolu sur la carte graphique, sinon uniquement sa construction sera possible.

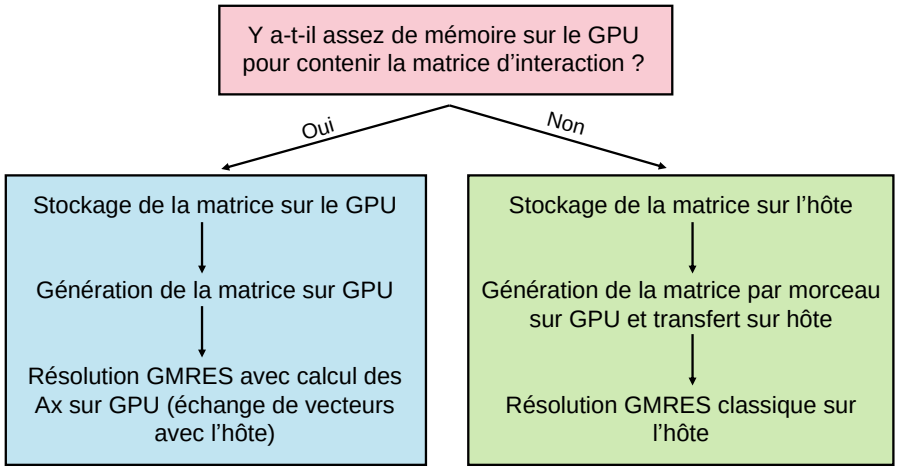


Figure 20 : Stratégie de calcul adaptative en fonction de la quantité de mémoire graphique disponible.

La Figure 21 présente les temps de construction de la matrice d'interaction en fonction de la méthode d'intégration (analytique ou numérique corrigée) sur différentes architectures. Le problème est le Cas20000. Le calcul est effectué en simple précision. Le graphique montre qu'il y a un ordre de grandeur sur les temps de calcul de différence entre chaque architecture. Même une carte graphique d'entrée de gamme, en comptant le temps de transfert de la matrice vers l'hôte, permet des calculs plus rapides d'un facteur 10 par rapport à un processeur monocoeur classique.

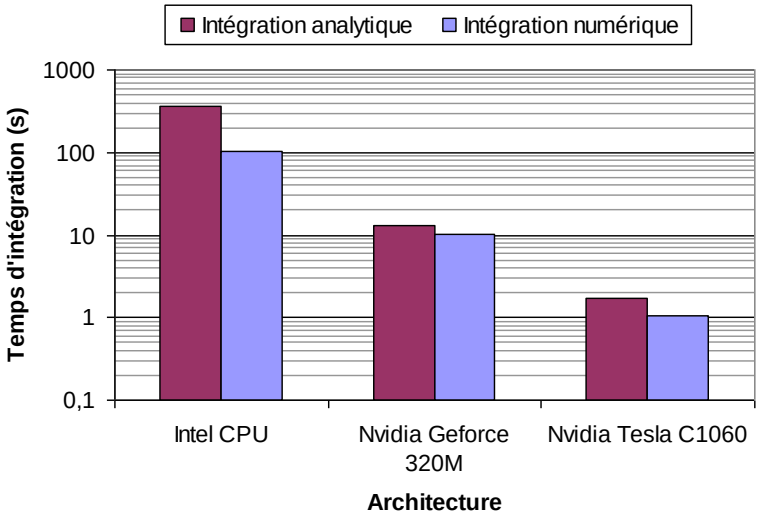


Figure 21 : Temps de construction de la matrice d'interaction en fonction de la méthode d'intégration sur différentes architectures. Le problème est le Cas20000.

Notons qu'il existe une méthode pour réduire les temps de transfert entre le GPU et l'hôte, c'est le streaming. Le principe est de transférer une portion de la matrice en même temps qu'une autre est calculée, ainsi les temps de calcul et de transfert ne sont pas cumulés. Nous avons testé cette méthode et les temps de calcul de la matrice et de son transfert sont presque divisés par deux. Malheureusement, cette technique requiert un format particulier en C de la matrice hors depuis Java l'accès à des tables en C est peu performant. Finalement, les gains obtenus sur les temps de transfert de la matrice sont perdus plusieurs fois lors des accès à la matrice au cours de la résolution itérative. Cette technique est souvent utilisée en C/C++ [Topa 11] mais en Java à travers JCuda elle ne s'est pas montrée efficace.

6 Résolution itérative sur processeurs graphiques

6.1 Stratégie de parallélisation d'un solveur itératif sur GPU

Nous avons vu au chapitre précédent que la résolution itérative avec GMRES nécessite de calculer un résidu à chaque itération [Saad 86]. La matrice d'interaction est stockée dans la mémoire vive de la carte graphique, nous effectuons donc le calcul des résidus sur cette dernière. Cette opération est hautement parallélisable sur GPU et nous utilisons les méthodes disponibles dans la bibliothèque CuBlas.

Nous disposons d'un algorithme GMRES en Java performant. Nous pouvons dans un premier temps réaliser un solveur hybride CPU-GPU, c'est-à-dire que nous allons utiliser le solveur Java tel quel avec seulement les produits matrice-vecteur qui sont réalisés sur le GPU. Cette approche est peu invasive mais des échanges de données entre l'hôte et le GPU sont alors nécessaires à chaque itération (envoi du vecteur solution et réception du résidu). Ces échanges de données sont coûteux en temps. Une autre solution pourrait être de développer un algorithme GMRES entièrement GPU, ou plutôt effectuant le maximum d'opérations possibles sur le GPU afin d'accélérer les calculs intermédiaires et de limiter les échanges de données entre le GPU et l'hôte. Nous souhaitons évaluer l'intérêt de réaliser ce solveur GMRES entièrement GPU qui serrait une entreprise très chronophage.

Nous choisissons d'effectuer des tests sur un algorithme de gradient conjugué (GC) [Fletcher 76] car il est facilement parallélisable et il se code rapidement. Nous allons

confronter l'approche hybride et entièrement GPU sur le cas du GC. Un algorithme GC est à priori plus facilement parallélisable que le GMRES. Si l'approche entièrement GPU n'est pas satisfaisante pour le GC alors elle ne le sera pas non plus à priori pour le GMRES et nous adopterons donc l'approche hybride.

Nous développons donc deux méthodes de résolution pour le gradient conjugué : la première effectue toutes les opérations en Java hormis le calcul du résidu qui est réalisé sur le GPU, la seconde est entièrement développée via des outils GPU, aucun transfert volumineux de données n'est alors nécessaire (uniquement quelques produits scalaires et les normes des résidus pour le critère d'arrêt).

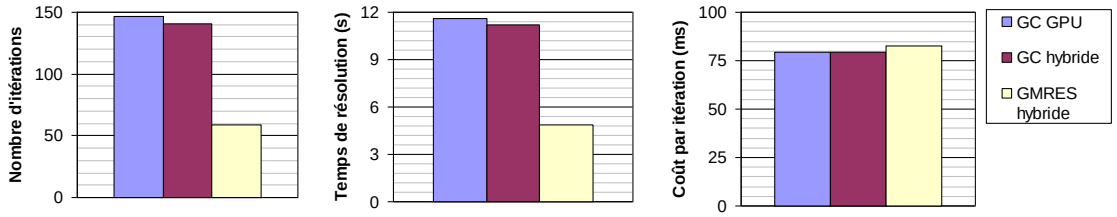


Figure 22 : Comparaison des performances des méthodes de résolution itérative sur GPU : gradient conjugué purement GPU, gradient conjugué hybride CPU-GPU et GMRES hybride CPU-GPU.

La Figure 22 présente une comparaison de différentes méthodes de résolution itérative sur GPU. La résolution par le GC hybride CPU-GPU est légèrement plus efficace que la version entièrement GPU. Notons qu'il y a quelques itérations de plus dans le cas du GC entièrement GPU. Cette différence s'explique par le fait que les opérations (hors calcul du résidu) sont effectuées en simple précision sur le GPU et en double précision sur l'ordinateur hôte.

Le coût par itération entre les deux GC est équivalent, ce qui peut sembler étonnant. En effet l'approche hybride est handicapée par les transferts de données entre le GPU et l'hôte et par l'utilisation d'un seul cœur de calcul (contre 240 pour l'approche GPU), elle devrait donc être moins efficace que l'approche GPU. La contre-performance de l'approche GPU peut s'expliquer par les opérations de réduction très coûteuses (voir chapitre I, 2.3) nécessaires au calcul des normes et des produits scalaires dans l'algorithme du GC. En conclusion, les performances d'une méthode de résolution itérative entièrement exécutée sur la carte graphique semblent très incertaines, il est donc préférable d'utiliser les méthodes hybrides CPU-GPU malgré les échanges de données entre l'hôte et le GPU.

Nous utiliserons donc le GMRES Java dont nous disposons couplé avec JCuda pour les calculs des résidus sur GPU. Cette solution a de plus l'avantage d'être non invasive. Une dernière remarque, la Figure 22 montre que le GMRES est plus efficace que le GC (sans préconditionneur).

6.2 Résolution du problème intégral

Nous présentons à la Figure 23 les temps de résolution du système d'équations en fonction du nombre de degrés de liberté pour différentes architectures. Les temps CPU sont comparés aux temps GPU en simple et en double précision.

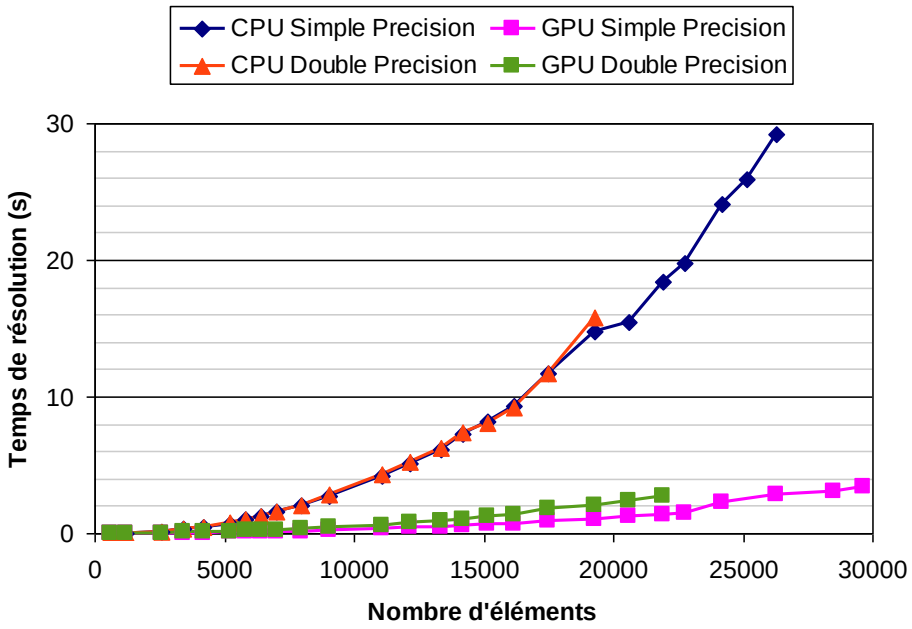


Figure 23 : Temps de résolution du système d'équations en fonction du nombre d'éléments. L'intégration est numérique avec corrections analytiques.

Le système d'équations est construit par intégration numérique corrigée. Le critère de convergence de GMRES est fixé à 1e-9. Une première observation est l'allure parabolique des courbes, elle est due aux produits matrice-vecteur nécessaires aux calculs des résidus. Deuxièmement, il n'y a pas de différence entre le simple et le double précision sur CPU. L'explication est que les opérateurs arithmétiques du Java sont nativement en double précision. Sur GPU, la résolution en simple précision est environ 2 fois plus rapide qu'en double (sur le Cas20000, 1,2s en simple, 2,4s en double). La raison de cette si faible

différence (en théorie un facteur 8, voir Figure 16) provient des transferts de données entre le GPU et l'hôte qui représentent une part importante du temps de résolution.

Nous présentons à la Figure 24 les accélérations de la résolution entre CPU et GPU en fonction du nombre d'éléments, pour des calculs en simple et en double précision. Comme pour l'intégration de la matrice d'interaction, le GPGPU est bien plus intéressant en simple précision, nous gagnons un ordre de grandeur. La résolution en double précision ne permet pas un gain d'un ordre de grandeur, le GPGPU n'est pas compétitif par rapport à une architecture multi-CPU dans ces conditions.

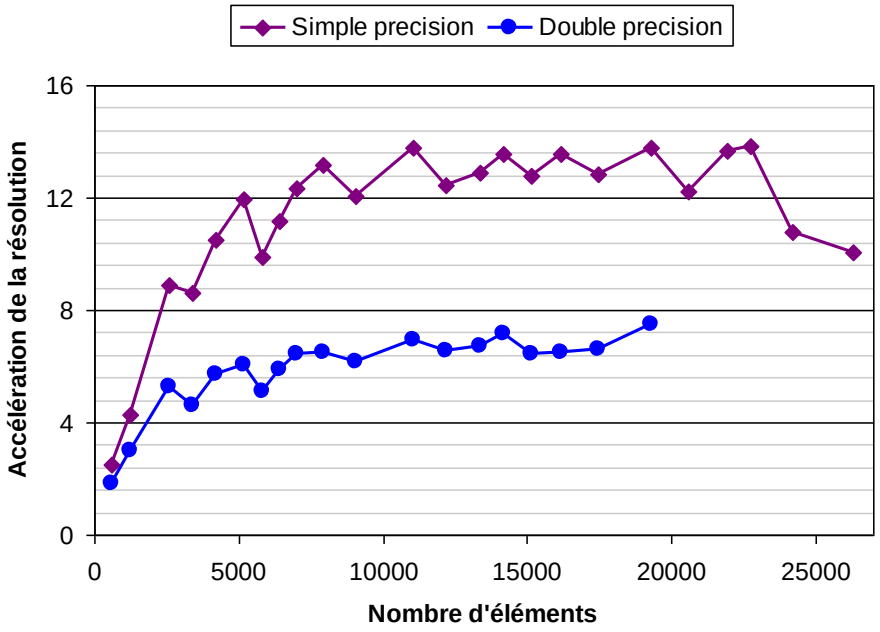


Figure 24 : Accélération de la résolution itérative apportée par l'utilisation du GPU par rapport au CPU en fonction du nombre d'éléments.

7 Bilan sur la parallélisation de la formulation intégrale

7.1 Performances de l'architecture parallèle

Nous proposons un bilan comparatif des temps de calcul obtenus sur le Cas9000 sur les différentes architectures parallèles Figure 25. Nous sélectionnons les meilleurs résultats obtenus sur chaque architecture. La référence est le calcul sur CPU monocoeur. Le

meilleur temps de calcul avec le cluster de PCs a été obtenu avec 4 PCs en réseau soit 8 cœurs de calcul. L'architecture multicoeur la plus performante est celle à mémoire partagé avec 4 cœurs. Le calcul sur GPU est réalisé avec une carte dédiée au calcul scientifique qui possède 240 cœurs de calcul, notons que le calcul est effectué ici en simple précision.

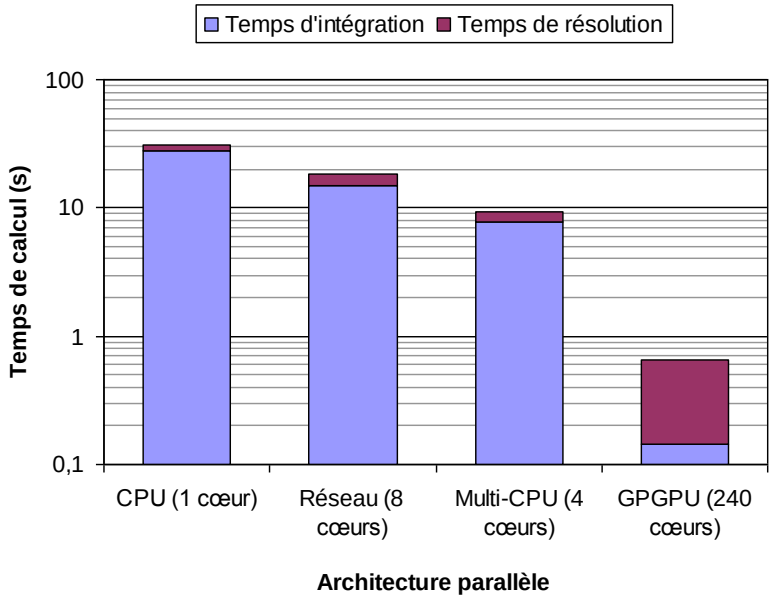


Figure 25 : Temps d'intégration et de résolution du Cas9000 sur les différentes architectures parallèles testées.

L'architecture la plus performante est le GPGPU, nous obtenons une accélération de près de 200 sur les temps de constructions de la matrice d'interaction. L'accélération totale en tenant compte de la résolution itérative est d'environ 50. La parallélisation sur processeur multicoeur offre des performances intéressantes compte tenu du nombre de cœurs, nous obtenons une accélération totale d'un peu plus de 3 pour 4 cœurs de calcul. L'expérience de calcul partagé sur un réseau n'a pas été concluante d'un point de vu accélération car le réseau n'est pas adapté à cet utilisation.

7.2 Discussions

Nous avons vu que la parallélisation apporte des gains en terme de performances de calcul. Supposons que le coût d'une modélisation numérique soit le coût du calcul en lui-même pondéré au coût en temps du développement du logiciel. Par exemple il n'est pas intéressant de consacrer une semaine à améliorer un logiciel pour obtenir des gains de

quelques pourcents seulement. Discutons pour chaque architecture du coût de sa mise en œuvre.

La parallélisation sur CPU multicoeur en architecture à mémoire partagée est très simple à mettre en place (multi-thread classique) et a montré une grande efficacité (scalabilité de 86% sur l'intégration de la matrice). C'est ce modèle qu'il faut développer en priorité car tous les ordinateurs actuels sont équipés de plusieurs cœurs de calcul.

La parallélisation sur un cluster de PC est plus complexe à mettre en œuvre car il est nécessaire de gérer explicitement les communications entre les tâches et les synchronisations. Il est également nécessaire de posséder le matériel adapté, en effet nous avons vu que dans le cas d'un réseau local classique les performances sont fortement dégradées par le coût des communications.

Le GPGPU s'est montré très efficace, une accélération de l'ordre de 300 a été obtenue. Cependant cette parallélisation nécessite une réécriture complète des codes pour prendre en compte les spécificités des GPU, et ce en langage bas niveau. La généricité des codes étant souvent une préoccupation majeure dans les équipes de développement, cette contrainte peut freiner l'adoption du GPGPU. De plus dans nos tests le GPGPU n'a réellement été efficace qu'en calcul en simple précision, cette limitation constitue un frein supplémentaire à son adoption car toutes les formulations ne restent pas valables en simple précision.

8 Conclusion

Nous avons, dans ce chapitre, parallélisé une formulation intégrale en potentiel en collocation à l'ordre 0. Les temps de calcul CPU ont été réduits grâce à l'utilisation de plusieurs processeurs. L'assemblage de la matrice d'interaction est très simple : à chaque terme de la matrice correspond une intégrale. La parallélisation est de ce fait très efficace.

Nous avons parallélisé les codes de calcul intégral sur des ordinateurs à processeurs multicoeurs : avec un processeur à 4 cœurs, l'accélération est d'environ 3,5 si la mémoire est partagée et 2,3 si la mémoire est distribuée. La différence s'expliquant par le coût des communications réseaux qui est très supérieur au coût des communications à travers un BUS. Nous avons également utilisé plusieurs ordinateurs en réseau afin d'en cumuler les ressources mémoires et processeurs. Les gains sur l'intégration atteignent rapidement leur

limite, les communications via un réseau local constituent un véritable goulot d'étranglement. Le réel intérêt d'un cluster de PCs est le cumul des ressources matérielles, notamment de la mémoire vive, qui permet de traiter des problèmes à nombre de degrés de liberté plus importants qu'il serait possible de faire sur une seule machine.

L'idéal semble d'avoir un grand nombre de processeurs sur une architecture à mémoire partagée. Le GPGPU est une solution, les cartes graphiques étant équipées de centaines de cœurs de calcul partageant une mémoire vive. Cependant les calculs ne sont vraiment qu'efficaces en simple précision, ce qui après vérification ne pose pas de problèmes pour notre formulation. Des programmes CUDA dédiés au calcul des intégrales ont été développés. La grille de calcul choisie associe un thread à chaque calcul d'intégrale. Des optimisations sur les transferts de données ont été mises en place. Les gains apportés sur le calcul de la matrice d'interaction sont d'environ deux ordres de grandeurs, ce qui est considérable. La résolution du système d'équations est effectuée par un algorithme itératif GMRES. Cette méthode nécessite le calcul de résidus qui sont minimisés jusqu'à l'obtention d'une solution de la précision souhaitée. C'est ce calcul du résidu qui est parallélisé. Sur les architectures à mémoire distribuées, les gains du parallélisme ne sont pas intéressants, en effet les communications réseaux sont très coûteuses. Sur GPGPU, ces gains sont d'un ordre de grandeur malgré les temps de transferts de données entre le GPU et l'hôte.

Le parallélisme, notamment sur GPGPU, permet de diminuer le coût en temps processeur du calcul de la matrice d'interaction et de sa résolution. Cependant, les besoins en mémoire vive nécessaire au stockage de la matrice limitent rapidement le nombre de degrés de libertés qu'il est possible de traiter. En effet, avec 4Go de mémoire il n'est pas possible de traiter des problèmes à plus de 30.000 éléments en simple précision. Nous proposons au chapitre suivant de réduire les besoins en mémoire vive en compressant la matrice par une décomposition en ondelettes.

9 Références

[Amdahl 67] G. Amdahl, « Validity of the single processor approach to achieving large scale computing capabilities », AFIPS spring joint computer conference, 1967.

[Buchau 08] A. Buchau, S.M. Tsafak, W. Hafla, W.M. Rucker, "Parallelization of a Fast Multipole Boundary Element Method with Cluster OpenMP," Magnetics, IEEE Transactions on , vol.44, no.6, pp.1338-1341, June 2008.

[D'Ambrosio 11] K. D'Ambrosio, R. Pirich, K. Petkov, A. Kaufman, "Method of Moments Software for GPU Hardware", Emerging Technologies for a Smarter World (CEWIT), 8th International Conference & Expo, 2011.

[Fletcher 76] R. Fletcher, « Conjugate gradient methods for indefinite systems », Lecture Notes in Mathematics, vol. 506, Numerical Analysis, pp. 73-89, 1976.

[Gustafson 88] J. Gustafson, « Reevaluating Amdahl's law », Communications of the ACM, May 1988.

[Lezar 10] E. Lezar and D.B. Davidson, « GPU Acceleration of Method of Moments Matrix Assembly using Rao-Wilton-Glisson Basis Functions », International Conference on Electronics and Information Engineering, 2010.

[Nvidia 11] Nvidia, « NVIDIA CUDA C Programming Guide », version 4, juin 2011.

[Peng 08] S. Peng and Z. Nie, « Acceleration of the Method of Moments Calculations by using Graphics Processing Units », IEEE Transactions on antennas and propagation, vol. 56, no 7, juillet 2008.

[Rubeck 11] C. Rubeck, O. Chadebec, B. Delinchant, J-P. Yonnet, and J-L. Coulomb, « JCuda vectorized and parallelized computation strategy for solving integral equations in electromagnetism on a standard personal computer », COMPUMAG 2011, Australie, 2011.

[Saad 86] Y. Saad and M.H. Schultz, « GMRES : A generalized minimal residual algorithm for solving nonsymmetric linear systems SIAM », Journal on Scientific and Statistical Computing, vol. 7, no 3, pp. 856-869, 1986.

[Topa 11] T. Topa, A. Noga and A. Karwowski, "Adapting MoM With RWG Basis Functions to GPU Technology Using CUDA", IEEE IEEE Transactions on antennas and propagation, vol. 10, 2011.

[Tracy 04] F. Tracy, « Optimizing finite element programs on the Cray X1 using coloring schemes, » *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium*, vol., no., pp. 313- 317, 7-11 June 2004.

Chapitre IV

Compression matricielle par ondelettes de la matrice d'interaction sur GPGPU

Sommaire

Résumé

Les méthodes intégrales génèrent des matrices qui sont en général pleines et par conséquent difficiles à stocker dans la mémoire vive. Nous proposons dans ce chapitre de réduire les besoins en mémoire en compressant la matrice d'interaction par ondelettes. Cette méthode a l'avantage d'être peu invasive, cependant elle reste de complexité parabolique. Nous utilisons alors le parallélisme sur GPGPU pour accélérer les calculs.

1 Introduction

Nous avons parallélisé une formulation intégrale au chapitre précédent, cependant nos problèmes sont limités à 30.000 degrés de liberté avec 4 Go de mémoire vive en simple précision. Il est nécessaire de réduire les besoins en mémoire vive dus au stockage de la matrice d'interaction. Les Méthodes Multipolaires Rapides (ou Fast Multipole Methods ou encore FMM) sont des méthodes très utilisées pour accélérer le calcul des matrices d'interaction des méthodes intégrales et limiter la quantité de mémoire nécessaire à leur stockage. Cependant, implanter une méthode FMM dans un code est souvent très invasif et la parallélisation peut s'avérer décevante. Nous proposons de compresser la matrice d'interaction par ondelettes, c'est une méthode proche de la compression d'images. L'avantage de cette méthode est qu'elle ne nécessite pas d'intervention au niveau de l'assemblage de la matrice d'interaction. L'inconvénient est que toute la matrice doit être calculée, la complexité de la construction du système d'équations est donc toujours parabolique. Nous utilisons alors le parallélisme sur architecture GPGPU pour accélérer les calculs, en effet la transformée en ondelettes est massivement parallélisable. Nous tentons ensuite une optimisation de l'algorithme en le couplant avec la méthode des matrices hiérarchiques. Cette méthode permet un partitionnement de la matrice d'interactions en blocs homogènes via une analyse de la géométrie du maillage.

Nous commençons le chapitre par la présentation d'une expérience de portage des Méthodes Multipolaires Rapides sur GPGPU qui ne s'est pas montrée pleinement satisfaisante car l'algorithme est difficilement parallélisable. Nous présentons ensuite la transformée en ondelettes rapide, puis l'algorithme de compression matricielle qui en découle. Nous appliquons alors cette méthode de compression à notre formulation intégrale puis nous la parallélisons sur l'architecture GPGPU. Pour finir nous couplons la compression matricielle par ondelettes avec les matrices hiérarchiques, cette partie manquant de maturité nous choisissons de la présenter dans l'Annexe D.

2 Digression sur les Méthodes Multipolaires Rapides sur GPGPU

2.1 Principe général de la compression d'un problème intégral

Greengard publie des travaux dans les années 80 dans lesquels sont présentés des méthodes qui réduisent la complexité du calcul du potentiel électrique créé par un nuage de charges ponctuelles [Greengard 87]. Le potentiel *V* au point *j* créé par un ensemble de charges ponctuelles est donné par :

$$V(\vec{x}_j) = \frac{1}{4\pi\varepsilon_0} \sum_{i \neq j} \frac{q_i}{\left\| \vec{x}_j - \vec{x}_i \right\|} \tag{1}$$

Où $\vec{x}_i$ et la position de la particule *i* et q_i sa charge électrique. Le principe de la méthode est de séparer le champ proche du champ lointain :

$$V = V_{\text{proche}} + V_{\text{lointain}} \tag{2}$$

L'idée de Greengard est d'approximer le potentiel lointain par un développement en série qui est ensuite tronqué. Dans le cas des méthodes multipolaires rapides (FMM), les séries choisies sont basées sur des développements en harmoniques sphériques [Greengard 87]. Les distributions lointaines de charges sont alors approximées par des multipôles. La complexité totale d'une résolution itérative par FMM est *0(N log N)* [Greengard 89] contrairement à celle d'une méthode intégrale sans compression qui est de *0(N²)*.

2.2 Décomposition multipolaire des interactions

Soit $\Phi(P)$ une solution de l'équation de Laplace en coordonnées sphériques. Elle peut être exprimée en série dont les termes sont des harmoniques sphériques [Ardon 10] :

$$\Phi(P) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \left(L_n^m . r^n + \frac{M_n^m}{r^{n+1}} \right) . Y_n^m(\theta, \Phi) \tag{3}$$

Les termes M_n^m et L_n^m sont respectivement les moments de l'expansion multipolaire et de l'expansion locale. Ces décompositions sont basées sur des sphères de validité. Dans le

cas du potentiel lointain (évalué à l'extérieur de la sphère contenant les charges) les coefficients L_n^m sont nuls afin de satisfaire la décroissance de potentiel à l'infini. Inversement, à l'intérieur de la sphère (et charges situées à l'extérieur) ce sont les termes de l'expansion multipolaire qui doivent être nuls. Les termes $Y_n^m.r^n$ et Y_n^m / r^{n+1} sont des multipôles respectivement de degré *n* et *–(n+1)*. Les coefficients Y_n^m sont des harmoniques sphériques définis à l'aide des polynômes de Legendre tel que :

$$Y_n^m(\theta,\Phi) = \sqrt{\frac{(n-|m|)!}{(n+|m|)!}}.P_n^{|m|}\cos(\theta).e^{im\phi} \tag{4}$$

Où les fonctions P_n^m sont les fonctions propres des polynômes de Legendre définis par :

$$P_n^m(x) = (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}P_n(x) \tag{5}$$

Où $P_n(x)$ désigne le polynôme de Legendre de degré *n*.

Le potentiel est approximé en tronquant la série à un certain degré. En considérant les lois de décroissance des champs, tout ensemble de sources électrostatiques, quelque soit sa complexité, peut être approximé par une simple charge à condition d'être suffisamment loin. Si le champ est exprimé à une distance moindre, il convient simplement de rajouter des termes (dipôles, quadripôles, octopôles, …). Typiquement, dans notre implémentation des FMM, nous considérons que la série peut être tronquées au quadripôle avec une précision acceptable, si la distance est supérieure à huit fois le rayon de la sphère englobant les charges [Ardon 10].

2.3 Opérateurs de calcul

Nous présentons dans cette partie les opérateurs intervenants dans le calcul des interactions entre les charges sources et les cibles où le potentiel est évalué. L'idée dans la décomposition des interactions en séries est d'exprimer des termes qui sont communs à plusieurs interactions, ainsi les calculs sont factorisés.

Soit deux distributions de charges ponctuelles contenues dans deux sphères S_Q et S_O respectivement de centre Q et O (Figure 1). Nous calculons le potentiel électrique créé par ces charges au point P dans S_O. Nous présentons les principaux opérateurs qui sont utilisés pour réaliser une décomposition multipolaire du potentiel créé par une distribution de charges ponctuelles. Ces opérateurs sont tous issus de la théorie des harmoniques sphériques, ils sont présentés en détails dans le mémoire de thèse de Vincent Ardon [Ardon 10].

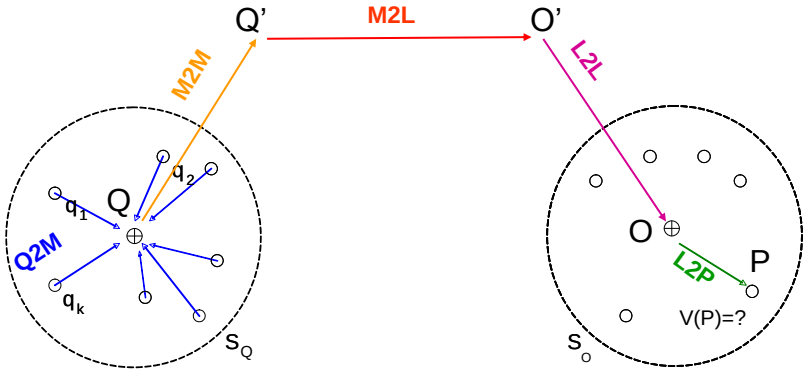


Figure 1 : Décomposition d'un chemin d'interaction entre un paquet de charges sources jusqu'à une cible où le potentiel est calculé.

***Q2M* : de la charge au multipôle**

L'opérateur *Q2M* permet de représenter la distribution des charges contenues dans S_Q en un multipôle équivalent au point Q.

***M2L* : du multipôle au local**

L'expansion multipolaire en Q est convertit en une expansion locale en O, c'est l'opérateur *M2L*.

***L2P* : du local au potentiel**

Les charges sont situées à l'extérieur de la sphère de validité S_O. L'opérateur *L2P* permet de calculer le potentiel au point P. Le chemin d'interaction complet *Q2M-M2L-L2P* permet alors de calculer le potentiel créé par les charges lointaines contenues dans S_Q au point P.

Translation des expansions multipolaires et locales

Dans la version multi-niveau des FMM, la décomposition à plusieurs niveaux des interactions nécessite des opérateurs de translation *M2M* et *L2L* des expansions

multipolaires et locales respectivement aux points *Q'* et *O'*. Le chemin d'interaction total s'écrit alors *Q2M-M2M-M2L-L2L-L2P*. Le calcul des opérateurs *M2L* représente le coût le plus important de l'implémentation de l'algorithme FMM. Cette opération est facilement parallélisable car elle est peu séquentielle contrairement aux translations, c'est donc ce calcul qui sera porté sur GPU.

Champ proche

Une dernière remarque, le potentiel total au point *P* dépend également des charges contenues dans S_O. Celles-ci sont traités en champ proche, c'est-à-dire classiquement en interaction totale.

2.4 Partitionnement de la géométrie avec un octree

Pour appliquer l'algorithme FMM sur une géométrie quelconque, il est nécessaire de regrouper les charges en paquets. Les relations d'éloignement entre ces paquets de charges définissent la nature des interactions (proches ou lointaines). De ces relations découlent également les opérateurs FMM.

Nous utilisons un octree pour effectuer ce partitionnement. Un octree est un découpage hiérarchique de la géométrie en cubes. Nous présentons à la Figure 2 le principe du partitionnement par un octree multi-niveau de niveau constant. Le critère qui définit dans quel cube est placé un élément est basé sur la position du barycentre. Le cube de niveau 1 contient toute la géométrie, il génère 8 cubes égaux de niveaux 2, etc. Nous obtenons alors 8^L cubes avec L le niveau de l'octree. Notons que les cubes vides sont ignorés. De plus, afin d'éviter d'avoir des cubes plus petits que les éléments ou de couper des éléments en deux, les algorithmes que nous possédons tiennent également compte du rayon des éléments.

Nous pouvons également utiliser un octree multi-niveau adaptatif. Le principe de partitionnement est identique, cependant les cubes qui ne contiennent pas un nombre minimum d'éléments ne sont pas subdivisés. Nous obtenons ainsi un nombre de charges par cube relativement homogène.

La géométrie étant partitionnée, nous pouvons à présent calculer les interactions. Les interactions entre deux cubes proches sont traitées classiquement en interactions totales.

Les interactions entre cubes éloignés sont traitées par FMM. Nous ne préciserons pas plus ici la notion de proximité ou d'éloignement car de nombreuses variantes de choix existent (cubes adjacents ou non, cubes « pères » adjacents ou non, etc…).

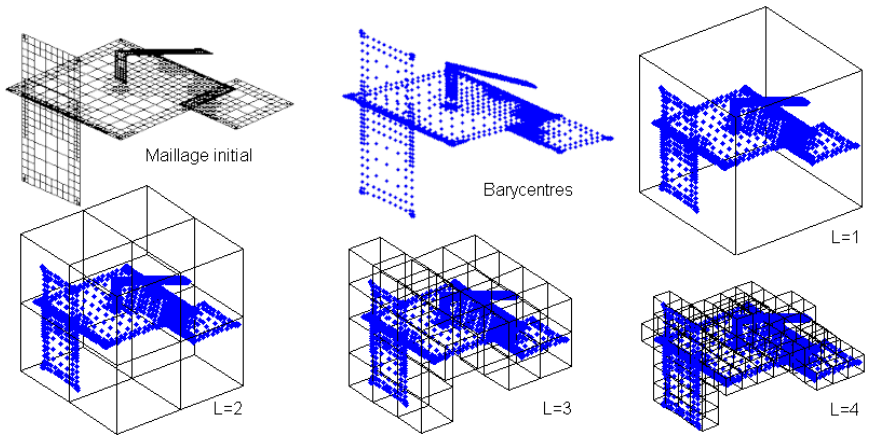


Figure 2 : Principe du partitionnement d'une géométrie par un octree de niveau L, critère sur les barycentres des éléments [Ardon 10].

2.5 Assemblage virtuel et résolution itérative

Nous parlons ici d'assemblage virtuel car la matrice d'interaction globale n'est pas construite explicitement. Cependant toutes les matrices d'interactions proches ainsi que les listes d'interactions et certains termes d'harmoniques sphériques nécessaires à la construction des opérateurs FMM peuvent être stockés si on ne désire pas refaire tous les calculs à chaque itération du solveur.

La matrice d'interaction n'existant pas ici, il n'est donc pas possible d'utiliser un solveur direct. Nous utilisons alors un solveur itératif GMRES [Saad 00]. Un vecteur de potentiel est calculé depuis un vecteur de charges donné à chaque itération. Le potentiel en champ proche est calculé classiquement via des produits matrice-vecteur. Le potentiel en champ lointain est calculé grâce aux opérateurs FMM.

2.6 Portage sur architecture GPGPU

Un portage de l'algorithme FMM a été effectué sur architecture GPGPU, les performances sont comparées avec une implémentation classique sur CPU en monocoeur. Ces travaux ont été réalisés principalement par Bertrand Bannwarth, ingénieur au G2ELab [Rubeck 11].

La première étape est l'analyse de la géométrie par un octree. De ce partitionnement découle une liste des chemins entre les sources et les cibles. Chaque chemin est exprimé en fonction des opérateurs présentés précédemment. Par nature, les calculs des chemins sont interdépendants, hiérarchisés et nécessitent des synchronisations. Ils sont de ce fait difficilement parallélisables.

L'approche choisie pour le portage GPGPU des FMM est de calculer uniquement les opérateurs *M2L* de chaque niveau sur GPU. En effet ce calcul représente le coût le plus important de l'algorithme FMM car ces opérateurs sont très nombreux et complexes à calculer. Pour un niveau donné, tous les coefficients des multipôles associés aux opérateurs *M2L* sont transférés dans la mémoire graphique. Chaque thread calcule un opérateur *M2L*. Les bases d'harmoniques sphériques sont calculées préalablement et stockées, elles ne dépendent que de la géométrie du problème et peuvent donc être réutilisées à chaque itération. Un kernel CUDA calcule à chaque itération une partie du produit matrice-vecteur pour le solveur GMRES.

Les matrices de champ proche sont construites avec le kernel CUDA présenté au chapitre III (5.2), elles sont ensuite transférées sur l'hôte pour réduire l'occupation mémoire du GPU.

Le cas test est le calcul de la capacité propre d'une sphère de 30000 éléments. Le CPU utilisé est un Intel Xeon cadencé à 2.67 GHz, la carte graphique utilisée est la Tesla C1060. Les calculs sont effectués en simple précision sur le GPU, nous avons vu au chapitre II (5.4) que la perte en précision des variables n'influençait pas le calcul de la distribution de charges électriques. De plus les FMM approximent le champ lointain, le passage à la simple précision ne devrait pas introduire d'avantage d'erreurs. L'octree est à multi-niveau adaptatif de niveau maximum 7, le degré des multipôles est de 3. La résolution itérative est réalisée par un algorithme GMRES dont le critère de convergence est réglé à 1e-9.

Nous présentons sur la Figure 3 une comparaison des temps de calcul entre l'approche CPU et GPGPU. Les deux grandes étapes de l'algorithme FMM sont la génération des tables d'interactions proches puis la résolution itérative FMM.

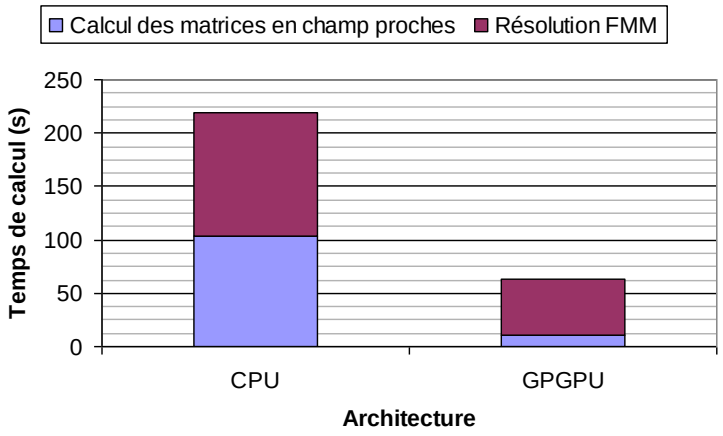


Figure 3 : Comparaisons des temps de calcul de certaines étapes de l'algorithme FMM entre les architectures CPU et GPGPU.

Les calculs en champ proche sont 10 fois plus rapides environ sur GPGPU que sur CPU. Ces résultats sont très en déca de ceux que nous avons obtenus au chapitre III (5.3), nous avions alors une accélération de deux ordres de grandeur entre le CPU et le GPU. Cette faible performance provient de plusieurs facteurs : tout d'abord les matrices sont plus petites que celles calculées au chapitre III et elles sont également très nombreuses d'où de très nombreux appels des routines CUDA qui n'effectuent à chaque fois que peu de calculs.

La durée globale du calcul n'est finalement accélérée que d'un facteur 3 grâce au GPGPU. Il devrait être possible d'améliorer le parallélisme de notre implémentation des FMM, mais cela nécessiterait une refonte en profondeur des codes, ce que nous n'avons pas envisagé dans le cadre de ces travaux, l'implémentation d'un algorithme du type FMM performant étant complexe et très chronophage.

2.7 Conclusion

La parallélisation de notre implémentation des FMM sur l'architecture GPGPU ne s'est pas montrée très efficace. La structure des données n'avait pas été prévue à cet usage au moment de l'écriture du code. Nous proposons alors de tester une méthode plus coûteuse en temps de calcul que les FMM mais qui n'est pas invasive et qui est parallélisable efficacement : c'est la compression matricielle par ondelettes.

3 Compression par ondelettes de la matrice d'interaction

3.1 Introduction à la transformation en ondelettes

Nous invitons le lecteur à consulter l'Annexe C qui présente des généralités de la transformation en ondelettes (TO). Nous utiliserons les propriétés de la TO pour construire un algorithme de compression matricielle.

3.1.1 Transformée en ondelette rapide

La transformée en ondelettes rapide (TOR) découle de l'analyse multi-résolution. L'idée générale est d'approximer une fonction de L^2 dans une suite croissante de sous espaces vectoriels de L^2 [Scheiblich 11]. Une dilatation permet de passer de la résolution 2^j, dans le sous espace V_j, à la résolution 2^{j-1}, dans le sous espace V_{j-1}. Les bases V_j sont construites par une fonction d'échelle Φ (ou ondelette père). Il existe une fonction Ψ appelée ondelette mère (ou ondelette analysante) qui crée une base W_j qui est le supplémentaire orthogonale de V_j, cet espace contient en fait la perte d'information (ou détails) entre les deux approximations successives V_j et V_{j-1} :

$$V_{j-1} = W_j \oplus V_j = W_j \oplus (W_{j+1} \oplus V_{j+1}) = \cdots, \quad W_j \perp V_j \tag{6}$$

La Figure 4 illustre la transformation d'un vecteur par une TOR. L'espace V_{j-1} est l'espace normal (V_0) qui est projeté au niveau j dans les sous-espaces V_j et W_j. La TOR construit les coefficients d'approximation V_j et d'ondelettes W_j uniquement pour les approximations V_{j-1}, c'est-à-dire que seuls les coefficients d'échelles sont décomposés. L'objectif est de filtrer les détails non significatifs entre chaque résolution, c'est le principe de la méthode de compression. Notons une conséquence importante de la TOR, le vecteur à transformer doit nécessairement être de dimension en puissance de deux.

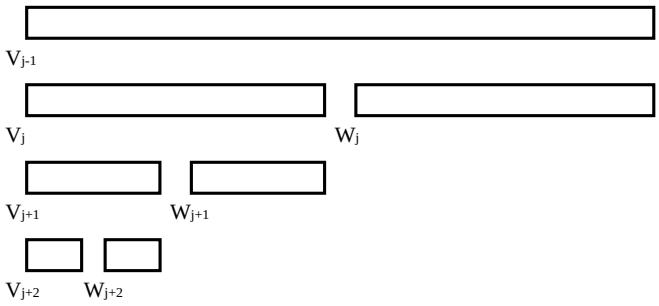


Figure 4 : Schéma de principe de la transformée en ondelettes rapide 1D.

3.1.2 Ondelettes de Haar et Daubechies D4

Le choix de l'ondelette est motivé par son orthonormalité qui permet d'effectuer des produits matriciels à la volée directement sur la matrice transformée et par son coût calculatoire. Nous utiliserons deux transformations en ondelettes : la transformée en ondelettes de Haar et la transformée en ondelettes de Daubechies D4 (voir Annexe C).

La transformation en ondelettes de Haar est donnée par :

$$W_{\perp}^{Haar} = \begin{bmatrix} c_0 & c_1 & & & & & & & & \\ d_0 & d_1 & & & & & & & & \\ & & c_0 & c_1 & & & & & & \\ & & d_0 & d_1 & & & & & & \\ & & & & \cdots & \cdots & \cdots & \cdots & & \\ & & & & \cdots & \cdots & \cdots & \cdots & & \\ & & & & & & c_0 & c_1 & & \\ & & & & & & d_0 & d_1 & & \\ & & & & & & & & c_0 & c_1 \\ & & & & & & & & d_0 & d_1 \end{bmatrix}_{2^j \times 2^j} \tag{7}$$

Où j est un entier supérieur ou égal à 1. Les coefficients c_k et d_k sont respectivement les coefficients d'échelles et d'ondelettes. Nous retrouvons bien ici la contrainte sur la taille du vecteur à transformer en ondelette, c'est-à-dire qu'elle soit en puissance de deux. Remarquons également que la taille minimale du vecteur éligible à la transformation est ici de deux. Les coefficients d'échelles et d'ondelettes associés sont :

$$\begin{aligned} c_0 &= 1/\sqrt{2} & d_0 &= 1/\sqrt{2} \\ c_1 &= 1/\sqrt{2} & d_1 &= -1/\sqrt{2} \end{aligned} \tag{8}$$

La transformation en ondelettes de Daubechies D4 est donnée par la matrice suivante [Ebrahimnejada 10] :

$$W_{\perp}^{DaubD4} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & & & & & & \\ d_0 & d_1 & d_2 & d_3 & & & & & & \\ & & c_0 & c_1 & c_2 & c_3 & & & & \\ & & d_0 & d_1 & d_2 & d_3 & & & & \\ & & & & \cdots & \cdots & \cdots & \cdots & & \\ & & & & \cdots & \cdots & \cdots & \cdots & & \\ & & & & & & c_0 & c_1 & c_2 & c_3 \\ & & & & & & d_0 & d_1 & d_2 & d_3 \\ c_2 & c_3 & & & & & & & c_0 & c_1 \\ d_2 & d_3 & & & & & & & d_0 & d_1 \end{bmatrix}_{2^j \times 2^j} \tag{9}$$

Où j est un entier supérieur ou égal à 2. Par conséquent la taille minimale du vecteur à transformer est de quatre.

Les coefficients d'échelles sont :

$$\begin{aligned} c_0 &= (1+\sqrt{3})/4\sqrt{2} \\ c_1 &= (3+\sqrt{3})/4\sqrt{2} \\ c_2 &= (3-\sqrt{3})/4\sqrt{2} \\ c_3 &= (1-\sqrt{3})/4\sqrt{2} \end{aligned} \tag{10}$$

Les coefficients de l'ondelette sont donnés par :

$$d_i = (-1)^i c_{3-i}, \quad i = 0,1,2,3 \tag{11}$$

3.1.3 Algorithme de la transformation en ondelettes rapide

La forme générale l'algorithme de la TOR est donnée par :

$$\begin{aligned} v_i^{j+1} &= \sum_{k=0}^{k=L} v^j_{(2i+k)\%(n/2^j+i)} c_k \\ v^{j+1}_{n/2^{j+1}+i} &= \sum_{k=0}^{k=L} v^j_{(2i+k)\%(n/2^j+i)} d_k \end{aligned} \tag{12}$$

Avec *v* le vecteur à transformer de longueur *n* et de résolution de niveau *j*. L'indice *i* varie de 1 à $n/2^{j+1}$ et *L* est le nombre des coefficients (ou longueur de l'ondelette).

Nous n'utilisons pas cette forme générique dans nos implémentations car la somme la rend peu performante. Nous avons implémenté des fonctions dédiées pour chaque ondelette dans lesquelles les sommes sont développées et les coefficients sont remplacés par leurs valeurs. L'opérateur modulo est éliminé, nous rappelons que c'est l'opérateur le plus coûteux en cycles d'horloge processeur. De plus notre implémentation est vectorisée.

3.1.4 Extension à la dimension 2

La transformée en ondelettes rapide est extensible à la dimension 2 (Figure 5) [Stollnitz 95]. Tout comme la transformation 1D, seuls les coefficients d'approximation sont projetés dans les sous-espaces de résolutions inférieures. Concrètement, la transformation en 2D est effectuée en appliquant la transformation 1D sur chaque ligne et ensuite sur chaque colonne (ou inversement, l'ordre dans lequel sont effectuées ces opérations n'a pas d'importance) [Ajdari 10].

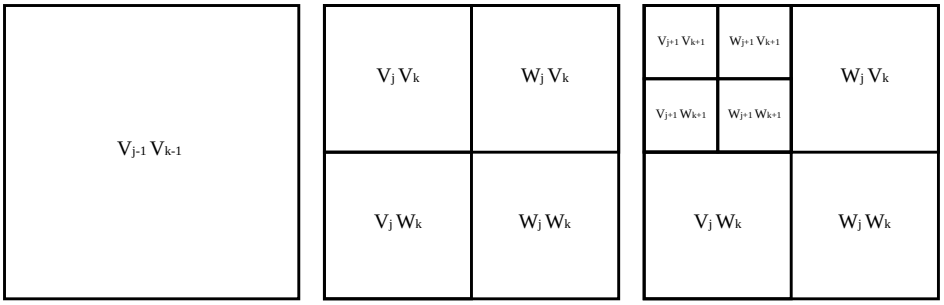


Figure 5 : Transformée en ondelettes rapide 2D.

3.2 Compression par ondelettes

Nous présentons ici un mini tutorial très didactique pour illustrer le principe de la compression par ondelettes à partir de la transformée en ondelettes rapide introduite précédemment [Mikulic 04].

Soit la transformation en ondelette de Haar non orthonormée suivante :

$$W^{Haar} = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{13}$$

Nous transformons un vecteur avec cette ondelette de Haar non orthonormée (Figure 6). Les éléments du vecteur sont projetés deux à deux dans deux nouvelles bases. Dans la base en vert, les nombres sont approximés deux à deux par leur valeur moyenne (fonction d'échelle), c'est le changement de résolution. La base en bleu contient la perte d'information lors du changement de résolution, ici les différences entre les nombres et leurs moyennes (fonction mère de l'ondelette). Nous avons vu que la transformation en ondelettes rapide consiste à répéter l'opération sur la nouvelle résolution jusqu'à atteindre la résolution minimale, ici il ne reste qu'un seul nombre. Ce nombre est d'ailleurs la valeur moyenne de tous les nombres du vecteur. La transformation est réversible, le vecteur d'origine se retrouve en augmentant successivement la résolution.

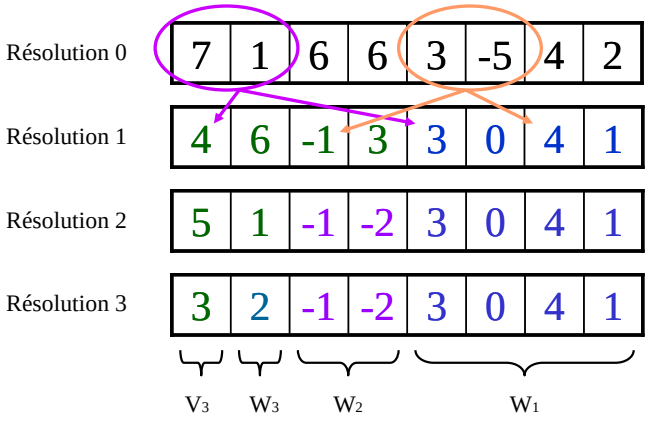


Figure 6 : Transformée discrète en ondelettes de Haar non orthonormées d'un vecteur.

La transformation en ondelettes n'est pas une méthode de compression en soit, en effet la transformée en ondelettes discrète transforme un vecteur de taille N en un autre vecteur de taille N. C'est l'apparition d'une quantité importante de zéros qui permet une réduction de la quantité de mémoire car ces derniers ne sont pas stockés si un format de matrice creuse est utilisé. Nous notons l'apparition d'un seul zéro lors de la transformation présentée sur la Figure 6, ce n'est pas suffisant et il est nécessaire d'en introduire d'avantage.

Soit le vecteur homogène présenté à la Figure 7, sa transformation en ondelettes de Haar génère un nombre très petit (relativement) dans la base des différences. L'opération de seuillage consiste à supprimer ce nombre, et ainsi à introduire un zéro supplémentaire. En d'autres mots, si la différence entre deux nombres consécutifs dans le vecteur est trop

faible, nous considérons que les deux nombres sont égaux et valent leur moyenne. C'est ce que nous voyons en deuxième partie de la figure, lors de la transformation inverse du vecteur. Nous constatons que la différence avec le vecteur d'origine reste faible relativement. De plus la valeur moyenne des éléments du vecteur reste inchangée.

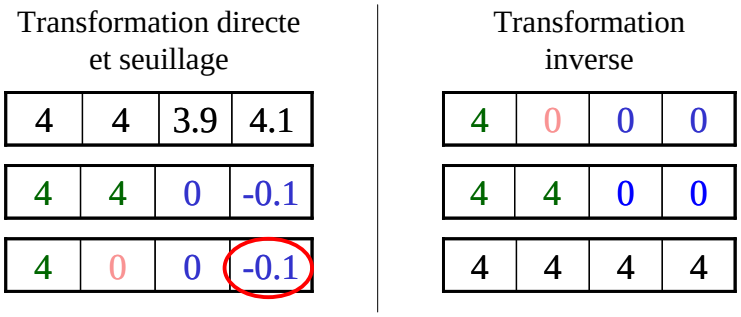


Figure 7 : Illustration de l'effet du seuillage sur un vecteur compressé par ondelettes.

Cette approche très simple illustre le principe de la compression par ondelettes qui consiste à introduire des zéros sur la transformée en ondelettes de la matrice d'interaction par une opération de seuillage. Une difficulté majeure va être de déterminer un paramètre de seuillage qui préserve suffisamment la matrice d'interaction afin de ne pas nuire à la qualité de la résolution du problème intégral.

3.3 Renumérotation des éléments du maillage

Dans le cas idéal, une matrice d'interaction générée par notre formulation électrostatique contient les valeurs les plus fortes aux alentours de sa diagonale (car en 1/r). Dans le cas d'une géométrie quelconque, rien ne nous assure que les valeurs les plus fortes se situent au voisinage de la diagonale. Nous utilisons alors un octree sur la géométrie qui renumérote les éléments en fonction de leur position. Ainsi, une proximité dans la numérotation des éléments se traduit également par une proximité spatiale dans le maillage. Les interactions entre des éléments proches se retrouvent bien au voisinage de la matrice d'interaction. Par ailleurs, la renumérotation des éléments par un octree favorise également la compression en créant des zones homogènes dans la matrice d'interaction.

3.4 Partitionnement en blocs de la matrice d'interaction

La matrice d'interaction, ne pouvant être calculée entièrement car elle n'est pas stockable dans la mémoire vive, est calculée par morceaux que nous appelons des blocs. L'ensemble des blocs est généré par une opération de partitionnement.

Nous avons vu que la transformée en ondelettes rapide nécessite des blocs de taille en 2^p, avec p un entier choisi par l'utilisateur. Pour un nombre de degré de liberté N donné, nous avons la relation [Scheiblich 09] :

$$N = n \cdot 2^p + R, \quad R < 2^p \tag{14}$$

Où n est un entier positif, le reste R est décomposé en nombre binaire :

$$R = \sum_{q=0}^{p-1} z_q \cdot 2^q, \quad z_q = \{0,1\} \tag{15}$$

La matrice décomposée en blocs est illustrée sur la Figure 8. Le paramètre p est choisi de façon à ce qu'un bloc de taille 2^px2^p ne sature pas la mémoire vive de l'ordinateur. Nous raffinons les blocs sur la diagonale (hachure), et nous choisissons de ne pas les compresser car la qualité de la solution obtenue par notre méthode intégrale dépend fortement de la diagonale (valeurs les plus fortes dans le cas idéal car en 1/r). De plus les blocs diagonaux vont également pouvoir être utilisés pour construire un préconditionneur par blocs diagonaux inversés [Ardon 10].

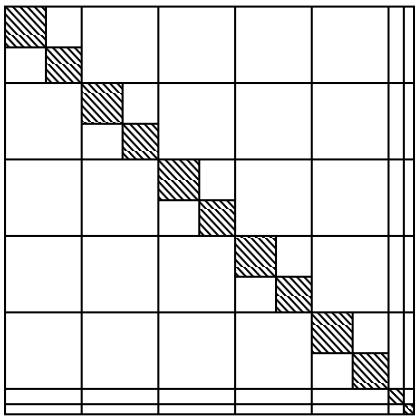


Figure 8 : Partitionnement de la matrice d'interaction en blocs de taille en puissance de deux. Raffinage de la diagonale.

3.5 Assemblage virtuel

Nous parlons ici d'assemblage virtuel [Bucher 04] car comme pour les FMM la matrice d'interaction globale n'est pas construite, elle est représentée par une liste de sous matrices compressées ou pas. Ces sous matrices peuvent être abstraites, la seule exigence est qu'elles puissent effectuer le produit matrice vecteur nécessaire à la résolution itérative. En programmation objet, les blocs contiennent la méthode de calcul de la sous matrice, la méthode de compression (ondelettes, etc.) et enfin la méthode pour effectuer le produit matrice vecteur. Une fois le partitionnement effectué et la liste des blocs créés, chaque bloc est calculé, puis compressé. Il est ensuite stocké en attente de la résolution itérative.

3.6 Seuillage de la matrice transformée par ondelettes

Le bloc calculé puis transformé par ondelettes doit être seuillé afin d'introduire les zéros nécessaires à sa compression [Bucher 03] :

$$S(x) = \begin{cases} x, & \text{si} |x| \geq \eta \\ 0, & \text{sinon} \end{cases} \tag{16}$$

Où *S(x)* est la fonction de seuillage appliqué à chaque terme de la matrice une fois transformée en ondelettes et η le seuil. Ce seuil doit être choisi de façon à ne pas dégrader significativement la matrice c'est-à-dire à conserver la qualité de la résolution du problème intégral.

Une approche très simple consiste à définir le seuil tel que [Scheiblich 09] :

$$\eta = \lambda \cdot \bar{x} \tag{17}$$

Où $\bar{x}$ est la valeur moyenne du bloc, et λ un réel supérieur à zéro, de préférence supérieur ou égal à 1. L'inconvénient est que ce paramètre dépend fortement de la géométrie, en d'autre terme il faut le calibrer pour chaque problème.

Koro et Abe proposent une méthode pour déterminer un seuil optimal qui n'introduit pas plus d'erreur que le maillage [Koro 03]. Cependant, l'estimation des erreurs nécessite des résolutions en amont avec des petits nombres de degrés de liberté. C'est également une

forme de calibration, un maillage à grand nombre de degrés de liberté doit être accompagné par deux sous maillages utilisés pour la détermination du seuil optimal.

Nous choisissons de développer une méthode plus stable que celle proposée par Scheiblich et moins contraignante que celle proposée par Koro. Elle est basée sur un critère de dégradation de la norme de la matrice. Rappelons que la transformée en ondelette orthonormée conserve la norme de la matrice (au sens de Frobenius). C'est un critère couramment utilisé en compression ACA[1] [Buchau 03, Rjasanow 07]. Voyons comment appliquer un tel critère à la compression matricielle par ondelettes.

La norme de Frobenius s'écrit :

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}\left|a_{ij}\right|^2} \tag{18}$$

C'est la norme 2 vectorielle appliquée à une matrice.

Nous cherchons pour une dégradation relative de la norme donnée, à déterminer la valeur du seuil à appliquer à la matrice. Par soucis de lisibilité nous nous placerons en 1D dans la démonstration.

Soit un vecteur *V* de *N* éléments tel que :

$$V = \{v_1, v_2, v_3, v_4, ..., v_N\} \tag{19}$$

Soit *n* sa norme 2, elle est donnée par :

$$n = \sqrt{\sum_{i}^{N} v_i^2} \tag{20}$$

Et soit $\tilde{n}$ la norme dégradée après le seuillage :

$$\tilde{n} = \sqrt{\sum_{i\neq\{k\}}^{N} v_i^2} \tag{21}$$

[1] Adaptative Cross Approximation : les blocs d'interactions lointaines sont compressés en les représentant par un produit de deux matrices de rangs inférieurs.

Les termes $v_{\{k\}}$ sont les termes négligés tel que :

$$\eta > v_{\{k\}} \tag{22}$$

Nous définissons alors la dégradation relative de la norme par :

$$\frac{n-\tilde{n}}{n} < \varepsilon \tag{23}$$

Où ε est le paramètre de dégradation de la norme. Essayons à présent de relier ce paramètre au paramètre de seuillage.

Tout d'abord écrivons la relation (23) sous une autre forme :

$$\left(\frac{n-\tilde{n}}{n}\right)^2 = \frac{n^2 - 2n\tilde{n} + \tilde{n}^2}{n^2} \tag{24}$$

Les deux normes *n* et $\tilde{n}$ étant très semblables relativement, faisons l'approximation suivante :

$$n\tilde{n} \approx \tilde{n}^2 \tag{25}$$

Nous pouvons alors écrire une forme équivalente à (23) :

$$\frac{n^2 - \tilde{n}^2}{n^2} < \varepsilon^2 \tag{26}$$

Revenons à la relation (22), et portons la au carré :

$$\eta^2 > v_{\{k\}}^2 \tag{27}$$

Faisons à présent la somme de tous ces termes seuillés :

$$\eta^2 \cdot N_k > \sum_{\{k\}} v_k^2 \tag{28}$$

Avec N_k le nombre de termes seuillés. Or nous ne connaissons pas le nombre de termes à seuiller, nous majorons alors la relation :

$$\eta^2 \cdot N > \eta^2 \cdot N_k > \sum_{\{k\}} v_k^2 \tag{29}$$

Nous ajoutons et retranchons $\tilde{n}^2$:

$$\eta^2 \cdot N > \sum_{\{k\}} v_k^2 + \sum_{i \neq \{k\}}^{N} v_i^2 - \sum_{i \neq \{k\}}^{N} v_i^2 \tag{30}$$

Nous obtenons :

$$\eta^2 \cdot N > n^2 - \tilde{n}^2 \tag{31}$$

Et nous trouvons finalement :

$$\frac{\eta^2 \cdot N}{n^2} > \frac{n^2 - \tilde{n}^2}{n^2} \tag{32}$$

Par identification des relations (26) et (32) nous lions les paramètres ε et η par :

$$\varepsilon^2 = \frac{\eta^2 \cdot N}{n^2} \tag{33}$$

Le paramètre de seuillage est alors donné par :

$$\eta = \sqrt{\frac{\varepsilon^2 n^2}{N}} \tag{34}$$

A notre connaissance l'utilisation d'un seuillage basé sur la dégradation relative de la norme des blocs est originale.

3.7 Stockage en matrice creuse

Une matrice possédant de nombreux éléments nuls ne nécessite pas moins d'espace de stockage qu'une matrice de dimension identique remplie de nombres quelconques. La réduction de la quantité de mémoire provient de l'utilisation d'un format matriciel adapté aux matrices creuses. Nous présentons ici deux méthodes couramment utilisées : le stockage matrice creuse au format coordonnée et au format ligne compressée.

Soit la matrice creuse de dimension $m_x n$ présentée à la Figure 9, elle est stockée classiquement dans un tableau à deux dimensions. Ce tableau stocke $m_x n$ nombres alors que seul N (ici 7) sont significatifs. L'espace mémoire qu'elle occupe en double précision est $8m_x n$, ici 25x8 = 200 octets.

0.0	0.0	6.0	0.0	0.0
0.0	0.0	0.0	3.0	4.0
7.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	9.0	0.0
0.0	1.0	0.0	2.0	0.0

Figure 9 : Matrice stockée classiquement dans un tableau à deux dimensions. Taille : 5x5x8 = 200 octets.

Le format de matrice creuse le plus simple à comprendre est le format par coordonnées (Figure 10). Trois tables sont utilisées : la première en double précision stocke les valeurs non nulles. Les deux suivantes sont des tables d'entiers qui stockent les coordonnées des valeurs non nulles de la matrice. L'espace mémoire utilisé est ici de $8N+4N+4N$, soit 7x(8+4+4) = 112 octets. Ce format utilise bien moins d'espace de stockage que la matrice pleine d'origine.

Val	6.0	3.0	4.0	7.0	9.0	1.0	2.0
Col	3	4	5	1	4	2	4
Lin	1	2	2	3	4	5	5

Figure 10 : Matrice creuse en format coordonnées. Taille : 7x(8+4+4)=112 octets.

Un des formats les plus couramment utilisé est le format par ligne (ou colonne) compressée (Figure 11). Il se présente également sous la forme de trois tables. Les deux premières sont identiques au format par coordonnées. La troisième exprime les coordonnées des lignes de manière compressée. Les nombres qu'elle contient sont les indexes des premiers éléments de chaque ligne. Ainsi la dernière table est de dimension le nombre de lignes de la matrice et non le nombre d'éléments non nuls. L'espace mémoire nécessaire ici est $8N+4N+4m$, soit ici 7x8+7x4+5x4 = 104 octets. Ce format est plus efficace que le format par coordonnées.

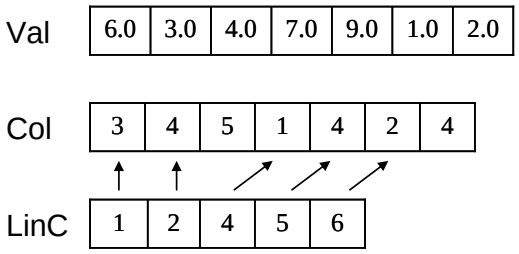


Figure 11 : Matrice creuse en format lignes compressées. Taille : 7x8+7x4+5x4=104 octets.

Dans nos problèmes, les sous matrices une fois transformées par ondelettes sont seuillées (donc perte en précision), par conséquent il est permis de stocker les nombres non nuls en simple précision sans introduire d'avantage d'erreurs [Bucher 04]. La quantité de mémoire nécessaire au stockage de la matrice creuse est encore diminuée : $4N+4N+4m$ donc ici 4x7+4x7+5x4 = 76 octets.

Il est important de souligner que les formats de stockage des matrices creuses sont intéressants seulement si le nombre de zéros est important. Par exemple au format par coordonnées, si la matrice précédente contenait 13 nombres non nuls alors elle occuperait 13x(8+4+4) soit 208 octets, c'est-à-dire d'avantage d'espace qu'en matrice pleine !

3.8 Produit matrice vecteur des blocs compressés par ondelettes

La résolution itérative GMRES [Saad 00] nécessite des produits matrice-vecteur de la forme :

$$A \cdot x = b \tag{35}$$

Où A est la matrice, x le vecteur contenant les inconnues et b le second membre. Cependant nous ne disposons pas de la matrice, mais de sa forme transformée en ondelettes.

Soit W la matrice de transformation en ondelettes. Si l'ondelette choisie est orthogonale, nous pouvons écrire [Sadiku 05] :

$$W^T = W^{-1}, \quad W^T W = I \tag{36}$$

Nous pouvons alors multiplier l'équation (35) par W :

$$W \cdot A \cdot (W^T \cdot W) \cdot x = W \cdot b \tag{37}$$

Ce qui donne :

$$(W \cdot A \cdot W^T) \cdot (W \cdot x) = W \cdot b \tag{38}$$

Les nouvelles matrices transformées sont définies par :

$$A^W = W \cdot A \cdot W^T \tag{39}$$

$$x^W = (W \cdot x) \tag{40}$$

$$b^W = W \cdot b \tag{41}$$

L'équation (35) peut alors s'écrire :

$$A^W \cdot x^W = b^W \tag{42}$$

Nous voyons que pour effectuer un produit matrice vecteur sur une matrice compressée, il est simplement nécessaire de transformer le vecteur à multiplier, effectuer la multiplication puis à transformer en inverse le résultat :

$$b = W^T b^W \tag{43}$$

Cette possibilité d'effectuer un produit matrice vecteur sans avoir à décompresser la matrice est la grande force de la méthode de compression matricielle par ondelettes.

3.9 Algorithme de compression matricielle de la matrice d'interaction

Récapitulons l'algorithme de compression matricielle de la matrice d'interaction d'une méthode intégrale. Nous avons toujours deux phases principales, la construction du système d'équations puis sa résolution [Bucher 04] :

Construction du système d'équations compressé

1. Application d'un octree et renumérotation des éléments
2. Partitionnement de la matrice d'interaction
3. Pour chaque bloc compressible
    3.1. Assemblage du bloc
    3.2. Transformation en ondelettes
    3.3. Seuillage

3.4. Stockage en format matrice creuse

4. Pour chaque bloc non compressible

4.1. Assemblage du bloc

4.2. Corrections analytiques (car bloc localisé sur la diagonale)

4.3. Stockage en format matrice pleine

Résolution itérative

1. Initialisation du solveur avec un vecteur solution initial
2. Tant que (critère de convergence)

2.1. Calcul du résidu

2.1.1. Pour chaque bloc compressé

2.1.1.1. Transformation directe du vecteur solution partiel associé au bloc

2.1.1.2. Calcul du produit matrice-vecteur sur le bloc compressé

2.1.1.3. Transformation inverse du résultat

2.1.1.4. Ajout du résidu partiel au résidu global

2.1.2. Pour chaque bloc non compressé

2.1.2.1. Calcul du produit matrice-vecteur

2.1.2.2. Ajout du résidu partiel au résidu global

2.2. Calcul de la nouvelle solution à partir du résidu

4 Application au calcul de capacités

4.1 Cas test

Le cas test est le condensateur plan du chapitre II (4.1). Il est constitué de deux plaques en vis-à-vis de dimensions 10x10 mm² et espacées de 2 mm. Nous calculons la capacité entre les deux plaques.

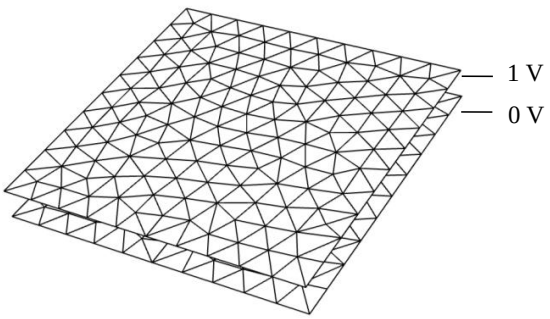


Figure 12 : Condensateur plan maillé en triangles.

Les plaques sont maillées en triangles, les éléments sont à l'ordre 0 et l'intégration est effectuée par la méthode de collocation (voir chapitre II, 3.2.3). Par défaut le nombre de liberté est 10940. Le partitionnement créé des blocs de taille en puissance de deux, les plus grands font au maximum 4096x4096, les blocs diagonaux font au maximum 256x256. Nous obtenons un total de 195 blocs et 2,3% d'interactions non compressées (blocs diagonaux). L'octree est de niveau 4. Le critère de convergence de GMRES est 1e-9 sans préconditionneur. Les calculs sont effectués en double précision.

Nous rappelons la configuration matérielle : le CPU est un Intel Xeon cadencé à 2.67 GHz en monocoeur et la carte graphique utilisée est une Tesla C1060 (240 cœurs, 4Go de mémoire). Les calculs sont effectués en simple précision sur le GPU.

Nous allons dans cette partie faire varier ces différents paramètres et étudier leurs influences principalement sur la qualité de la résolution (donnée par l'erreur relative de la capacité par rapport au problème sans compression) et le taux de compression. Le taux de compression est défini par le rapport entre l'occupation mémoire du système compressé et l'occupation mémoire théorique du système non compressé.

4.2 Validité du stockage en simple précision

Les données en format de matrice creuse sont stockées en simple précision, la matrice ayant déjà subi une opération de seuillage, cette perte en précision ne devrait pas avoir d'effet sur la résolution du problème intégral associé à notre formulation électrostatique [Bucher 04]. Nous proposons de le vérifier. Nous comparons deux résolutions absolument identiques (le cas par défaut décrit précédemment) hormis la précision des variables dans le stockage des matrices creuses (voir 3.7).

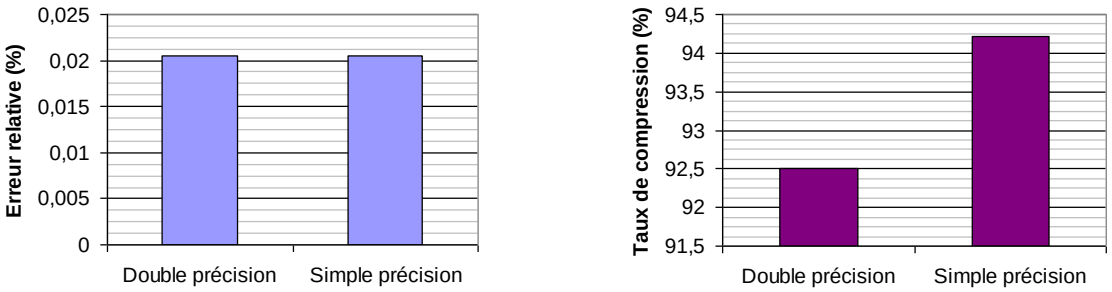


Figure 13 : Erreur relative sur la capacité et taux de compression en fonction de la précision des matrices creuses.

Nous voyons sur la Figure 13 que l'erreur relative sur la capacité n'est pas influencée (aucune différence à 8 chiffres significatifs) par la perte en précision du stockage des matrices creuses. Par contre, le taux de compression est lui amélioré par le passage en simple précision (près de 2% ici). Stocker les matrices creuses en simple précision est donc recommandé pour notre formulation électrostatique.

4.3 Pertinence de la méthode de seuillage

Nous avons développé un critère de seuillage basée sur la dégradation de la norme des matrices. Nous présentons sur la Figure 14 l'erreur relative sur la capacité en fonction de la dégradation relative de la norme des blocs. Nous comparons les résultats entre deux ondelettes, l'ondelette de Haar orthonormée et l'ondelette de Daubechies D4. Les deux courbes sont en décroissance (logarithmique), c'est-à-dire que plus les blocs sont dégradés et moins la capacité est précise. Nous notons également que la compression avec l'ondelette de Daubechies D4 est un peu plus précise qu'avec l'ondelette de Haar.

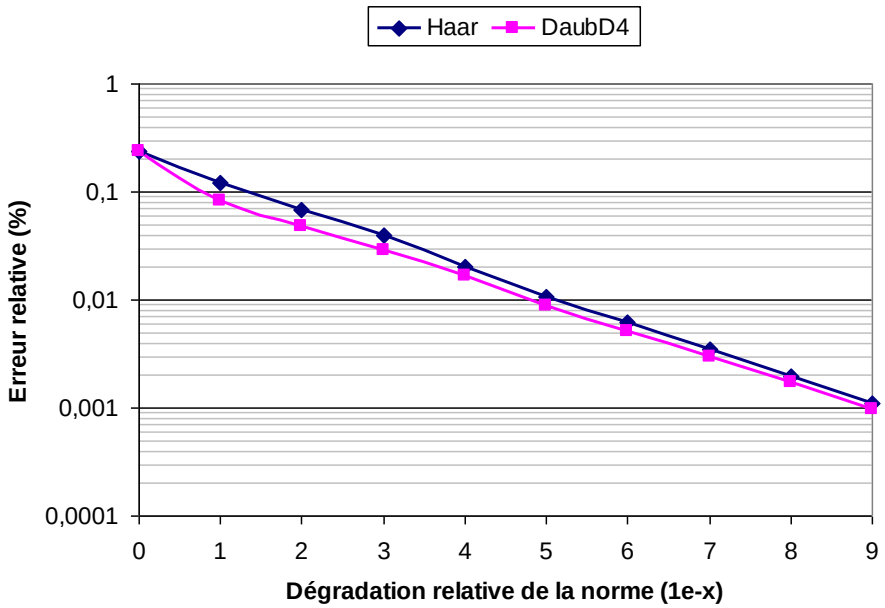


Figure 14 : Erreur relative de la capacité en fonction du critère de dégradation de la norme de la matrice d'interaction.

Voyons l'effet du critère de dégradation de la norme sur le taux de compression (Figure 15). Comme nous l'attendions, plus les blocs sont dégradés et plus ces derniers se creusent.

L'ondelette de Haar permet ici un taux de compression un petit peu plus élevé (de 2% pour une dégradation de la norme à 1e-9) qu'avec l'ondelette de Daubechies D4.

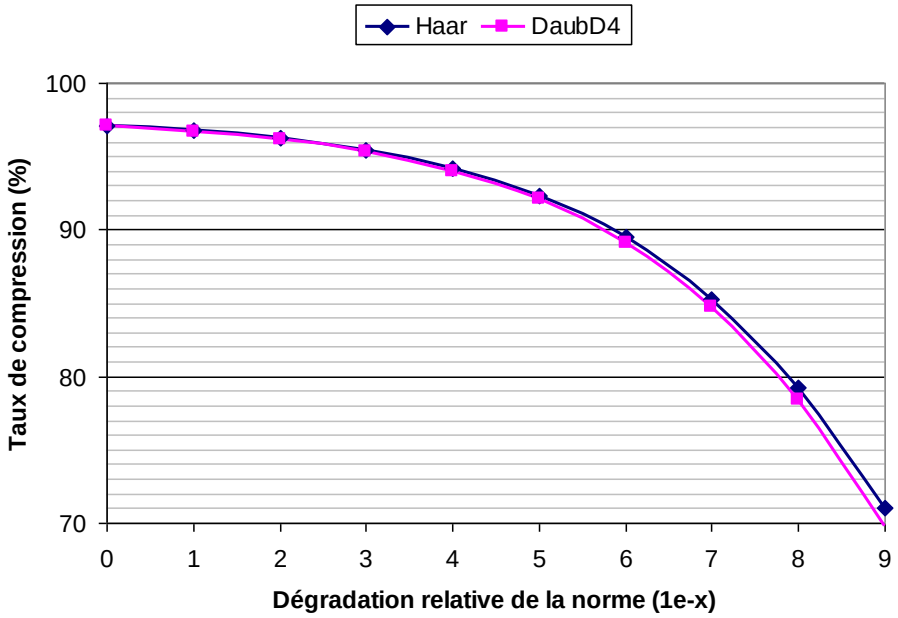


Figure 15 : Taux de compression en fonction du critère de dégradation de la norme de la matrice d'interaction.

4.4 Choix de l'ondelette

Nous avons vu que l'ondelette de Haar permet un meilleur taux de compression que l'ondelette de Daubechies D4, mais le résultat est moins précis qu'avec cette dernière. Quelle est alors l'ondelette qui permet le meilleur taux de compression pour une erreur donnée ? Nous répondons à cette question sur la Figure 16, nous traçons le taux de compression en fonction de l'erreur relative sur la capacité. L'ondelette la plus performante est l'ondelette de Daubechies D4. Cependant nous verrons que cette ondelette est plus coûteuse en temps de calcul et surtout qu'elle se parallélise assez mal. L'ondelette finalement privilégiée sera l'ondelette de Haar.

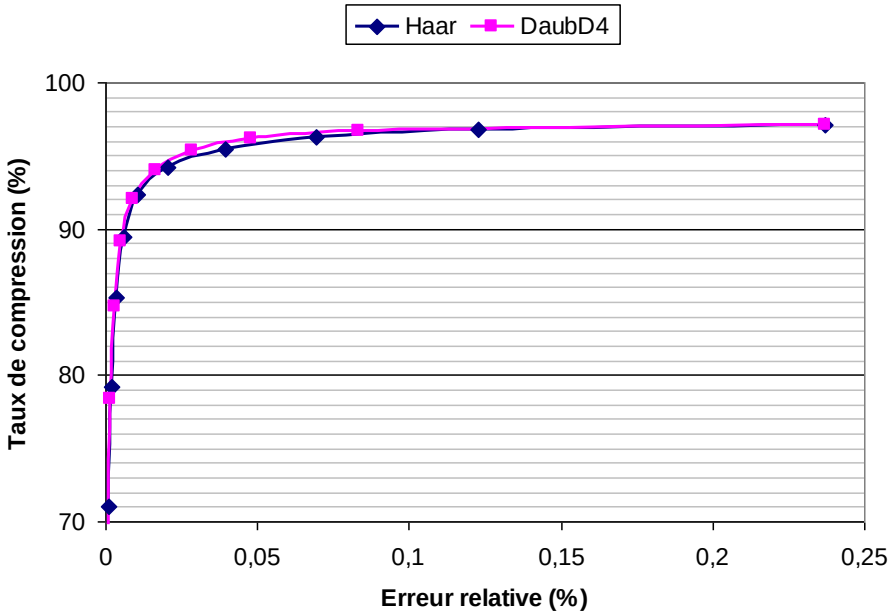


Figure 16 : Taux de compression en fonction de l'erreur relative sur la capacité.

4.5 Influence de la renumérotation des éléments avec un octree

Nous appliquons un octree à niveau constant sur la géométrie. Une renumérotation des éléments du maillage est effectuée afin de regrouper les interactions proches aux alentours de la diagonale de la matrice d'interaction. Cette renumérotation crée également des zones homogènes dans la matrice qui devrait favoriser la compression. C'est ce que nous vérifions sur la Figure 17 qui représente le taux de compression en fonction du niveau de l'octree. Nous vérifions que plus la renumérotation est fine et meilleure est le taux de compression. La corrélation est pratiquement linéaire. Le palier à partir du niveau 5 peut s'expliquer par le fait que les boites qui composent l'octree sont plus fines que les éléments du maillage, il n'a donc plus aucun effet.

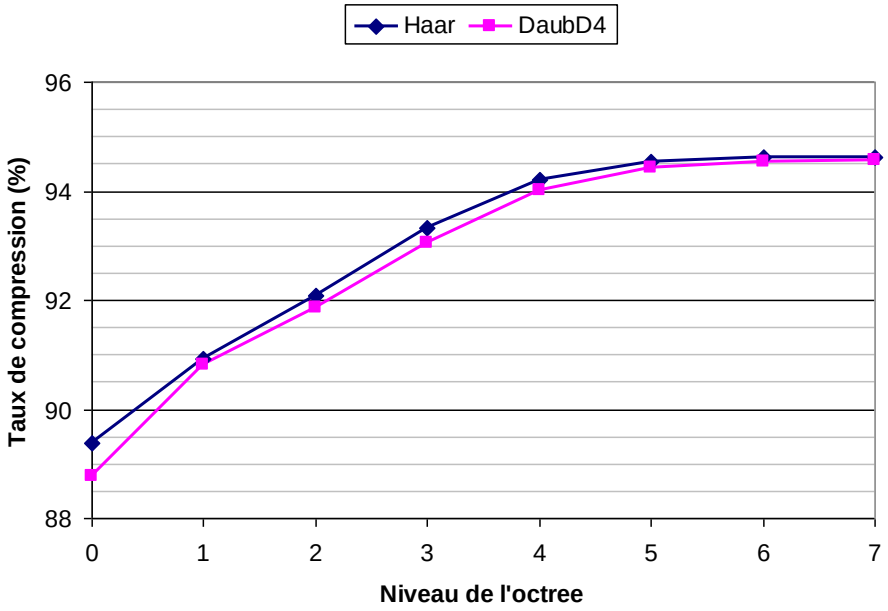


Figure 17 : Taux de compression en fonction du niveau de l'octree.

4.6 Compression des blocs diagonaux

Intéressons-nous à l'effet de la compression des blocs diagonaux. Ces derniers contenants les interactions les plus fortes, leur compression devrait être plus sensible sur la précision de la résolution. Nous proposons de compresser les blocs diagonaux sur le cas test par défaut. Nous faisons varier le paramètre de dégradation de la norme des blocs diagonaux uniquement, les autres blocs sont seuillés avec le paramètre par défaut (1e-4). La diagonale de la matrice est extraite est stockée à part, en effet si elle est altérée la résolution ne converge pas.

Nous présentons sur la Figure 18 l'erreur relative de la capacité en fonction de la dégradation des blocs diagonaux. Les courbes de références sont celles obtenues précédemment (compression des blocs sauf diagonaux). Cette fois ci les courbes ne sont pas linéaires (échelle logarithmique) : l'erreur est instable avec l'ondelette de Haar lorsque la dégradation relative de la norme est élevée (1e-0 à 1e-3). Il est toutefois possible d'atteindre les mêmes précisions que celles obtenues sans compression des blocs diagonaux (à partir de 1e-8).

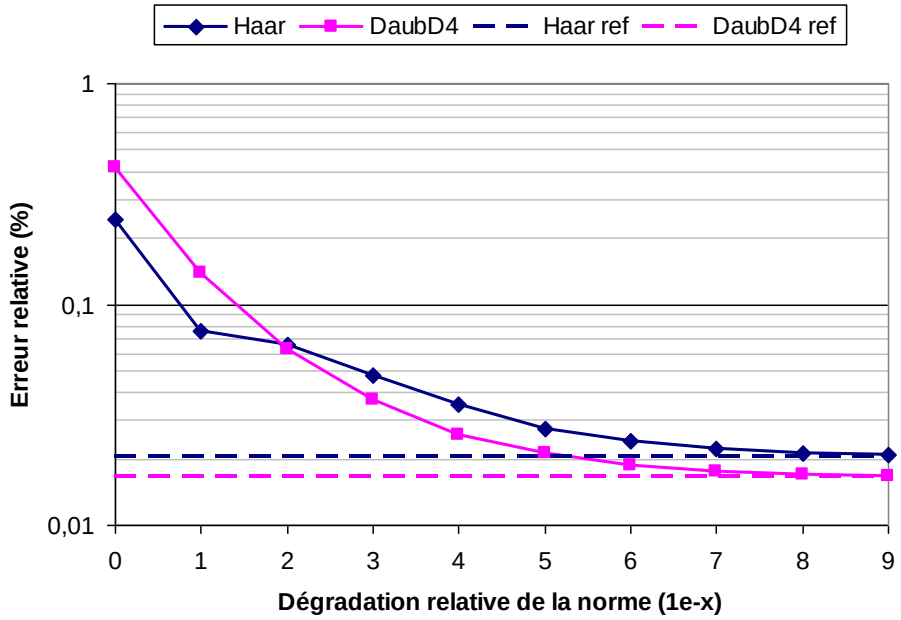


Figure 18 : Erreur relative de la capacité en fonction de la dégradation relative de la norme des blocs diagonaux.

La Figure 19 montre le taux de compression en fonction de la dégradation de la norme des blocs diagonaux.

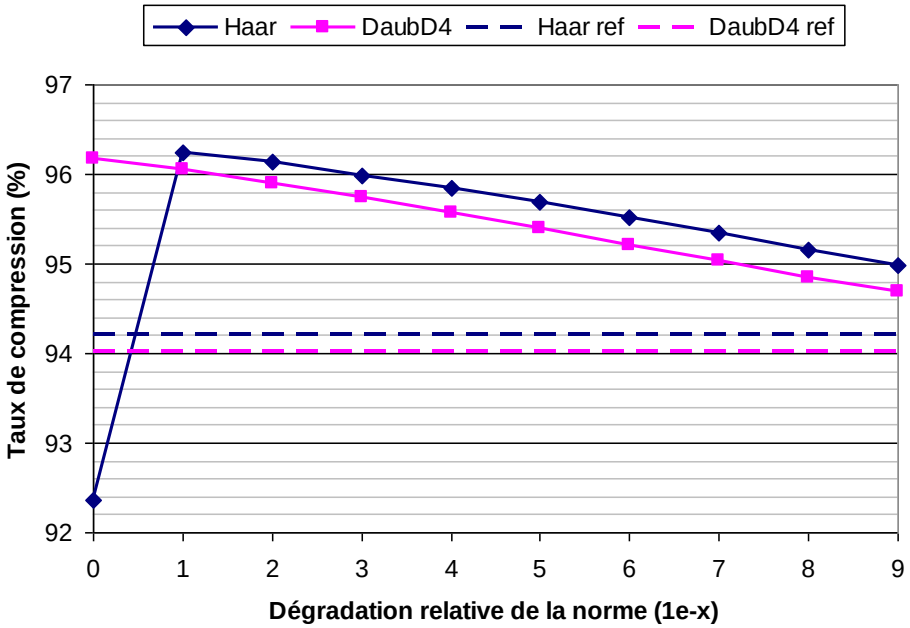


Figure 19 : Taux de compression en fonction de la dégradation relative de la norme de blocs diagonaux.

Dans notre exemple nous pouvons gagner près de 1 pourcent de compression sans nuire à la qualité de la résolution (dégradation 1e-8 à 1e-9). Ce chiffre dépend évidemment du pourcentage d'interactions situées dans les blocs diagonaux (environ 2% ici).

Il est donc possible de compresser les blocs diagonaux et de gagner de l'espace mémoire sans nuire à la qualité de la résolution (en imposant toutefois un paramètre de dégradation de la norme plus contraignant que sur le reste de la matrice). Nous ne le ferons pas car nous utiliserons les blocs diagonaux pour générer un préconditionnement de la matrice par blocs diagonaux inversés.

4.7 Préconditionnement de la matrice

Les solveurs itératifs sont très sensibles au conditionnement des matrices. Nous pouvons en général fortement améliorer la convergence par l'utilisation de préconditionneurs. Nous proposons de comparer sur la Figure 20 le nombre d'itérations nécessaires et les temps de calcul d'une résolution GMRES en fonction du préconditionneur utilisé. Le préconditionneur de Jacobi est obtenu en inversant les termes diagonaux de la matrice. Les blocs diagonaux inversés sont obtenus par une décomposition LU des blocs diagonaux.

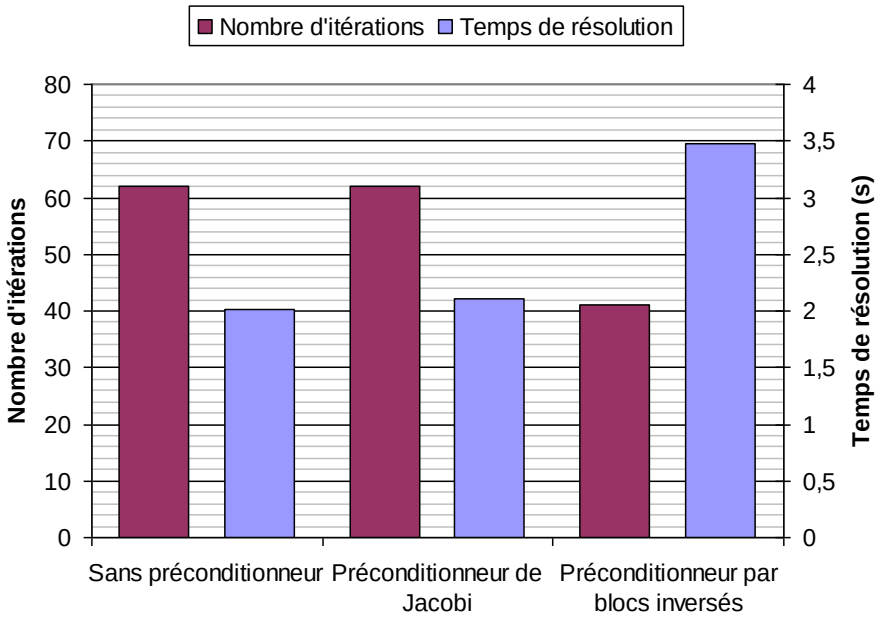


Figure 20 : Nombre d'itération et temps de résolution en fonction du préconditionneur.

Nous observons sur la Figure 20 que le préconditionneur de Jacobi n'améliore pas la convergence et coûte plus cher en temps de calcul. Le préconditionneur par blocs diagonaux inversés diminue le nombre d'itération de près de 30%. Cependant son coût calculatoire (inversion des blocs + application du préconditionneur) provoque une augmentation de près de 50% du temps de résolution. L'utilisation d'un préconditionneur n'est pas intéressant d'un point de vu temps de calcul pour notre cas test. Par contre, nous verrons au chapitre suivant que le préconditionneur par blocs inversés est indispensable pour faire converger la solution sur un problème de complexité industrielle.

4.8 Taux de compression et occupation mémoire

Nous traçons sur la Figure 21 le taux de compression en fonction du nombre d'éléments. Nous constatons que plus le nombre d'éléments est élevé, meilleure est la compression. Cela s'explique par le fait que plus la matrice d'interaction est grande, plus les zones homogènes quelle contient sont étendues et donc plus efficace est la compression. De plus, le pourcentage d'interactions proches diminue en fonction du nombre d'éléments car les blocs diagonaux non compressés ont une taille constante (ici 256x256).

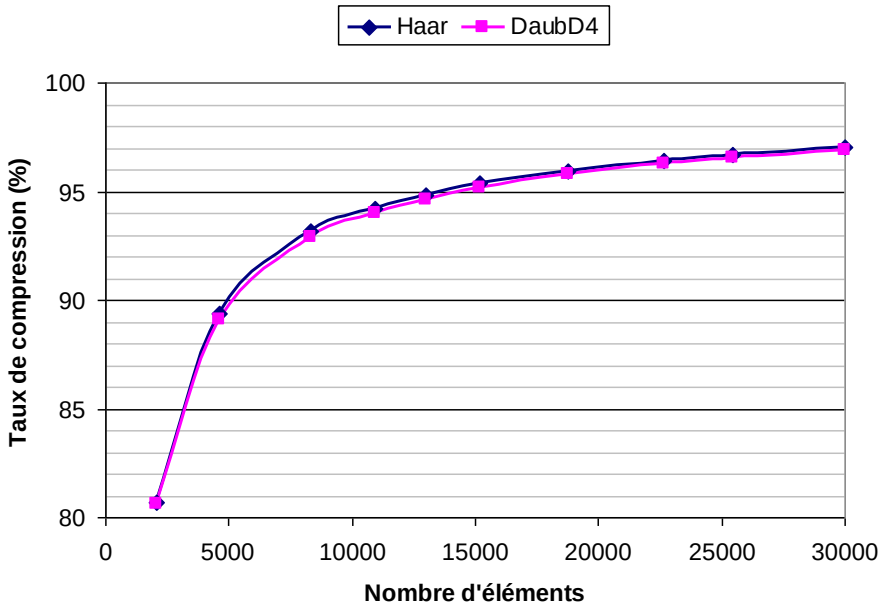


Figure 21 : Taux de compression en fonction du nombre d'éléments.

Intéressons-nous maintenant à l'occupation mémoire en fonction du nombre d'éléments (Figure 22). L'occupation mémoire du cas sans compression est parabolique, c'est ce que nous avons vu au chapitre II (3.2.3). Les occupations mémoire des cas compressés sont linéaires ! Cette caractéristique est classique des éléments finis ou des FMM [Greengard 87].

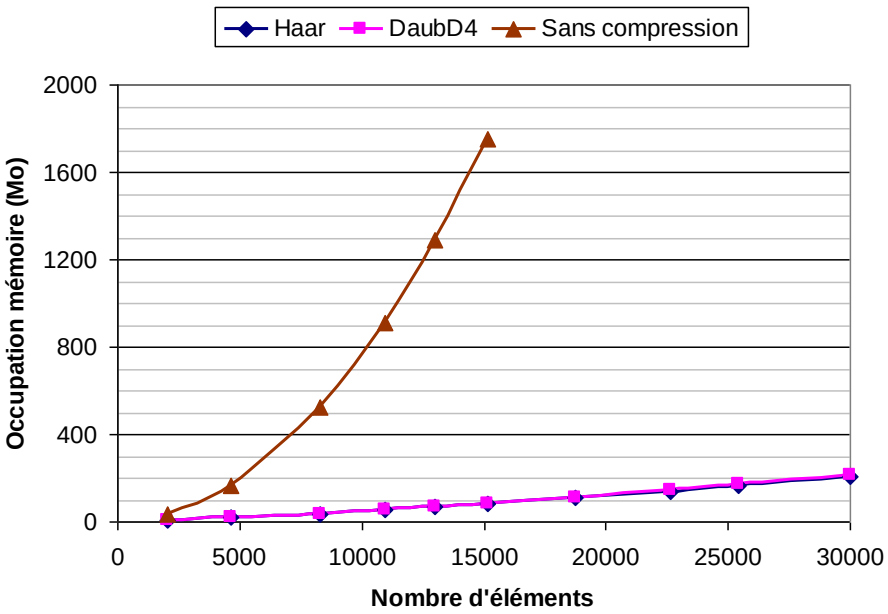


Figure 22 : Occupation mémoire en fonction du nombre d'éléments.

4.9 Temps d'intégration et de résolution

Nous présentons sur la Figure 23 les temps de construction du système d'équations en fonction du nombre d'éléments. Ils sont paraboliques dans tous les cas car le calcul de toutes les interactions est nécessaire. La compression ajoute un léger coût par rapport au problème non compressé (<10%). Nous remarquons également que l'utilisation de l'ondelette de Daubechies D4 est plus coûteuse que celle de Haar.

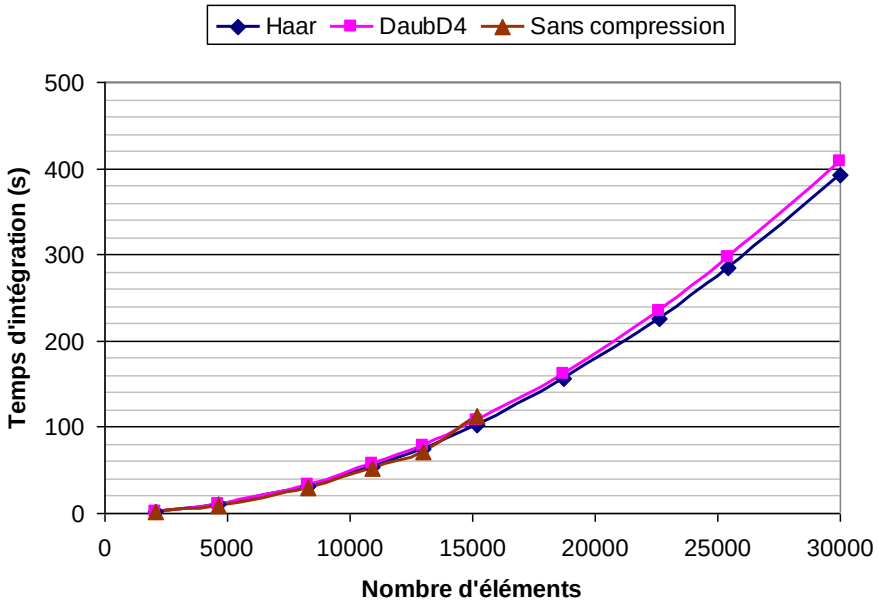


Figure 23 : Temps d'intégration et de compression en fonction du nombre d'éléments.

Les temps de résolution sont présentés sur la Figure 24. La complexité du cas non compressé est parabolique tandis qu'elle est en *O(N log N)* dans les cas compressés. Nous retrouvons cette caractéristique classique des éléments finis ou des FMM [Greengard 87].

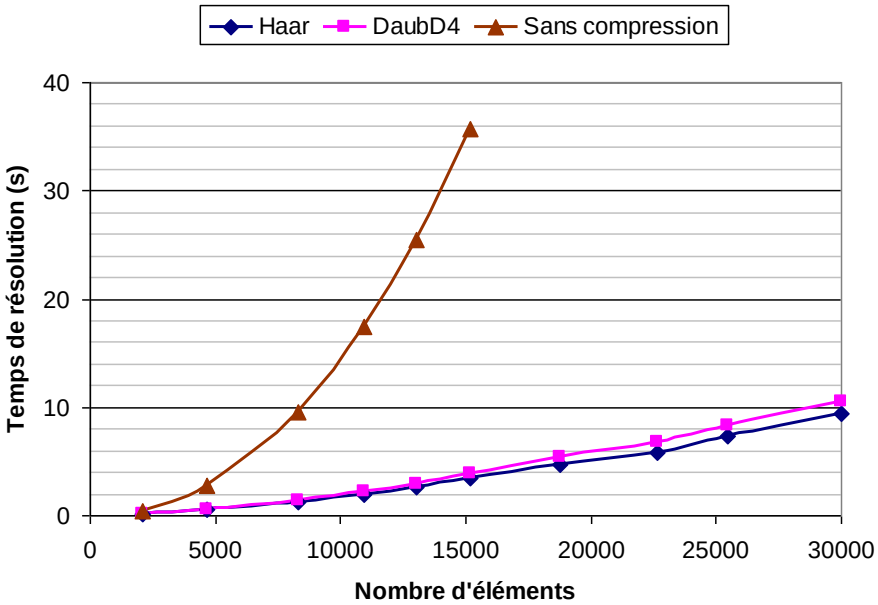


Figure 24 : Temps de résolution en fonction du nombre d'éléments.

5 Compression par ondelettes sur GPGPU

5.1 Transformation en ondelettes sur GPGPU

5.1.1 Parallélisation de la transformée en ondelettes rapide

Comme pour l'implémentation CPU, la transformation en ondelette est tout d'abord appliquée sur les lignes de la matrice. Un noyau CUDA dédié a été écrit pour chaque ondelette. La matrice est ensuite transposée puis la transformation est à nouveau appliquée. La matrice est transposée une deuxième fois pour retrouver son format d'origine. Il est en effet plus efficace d'appliquer la transformation en ondelettes sur les lignes de la matrice transposée que sur les colonnes car dans ce dernier cas nous perdons la coalescence des données (matrice stockée en format ligne) ce qui est très coûteux en temps de calcul [Jianjun 11].

Des kernels CUDA ont été développés pour effectuer la transformation en ondelettes rapide sur GPU. Nous présentons sur la Figure 25 celui correspondant à la transformation en ondelettes de Haar. La grille de calcul est en 2D, les blocks de threads sont en 1D suivant les lignes. Pour chaque block, une portion de ligne est extraite de la matrice dans la mémoire globale (l'accès est coalescent). Cette portion de ligne est alors projetée dans les différentes bases successives. Nous définissons *n/2* threads par block avec *n* la taille de la portion de ligne. Chaque thread calcule un coefficient d'échelle et un coefficient d'ondelette. Les coefficients d'ondelettes des bases W_j sont directement envoyés dans la matrice cible dans la mémoire globale, l'accès à la mémoire n'est pas optimal ici. Les coefficients d'échelle des bases V_i restent dans la mémoire partagée afin d'y effectuer la projection suivante jusqu'à la résolution maximale qu'il est possible d'atteindre. Nous voyons que la transformation n'est pas entièrement appliquée ici, l'approximation *j+2* n'est pas l'approximation maximale. Il est donc nécessaire d'appliquer le noyau CUDA une deuxième fois sur les données à transformer restantes.

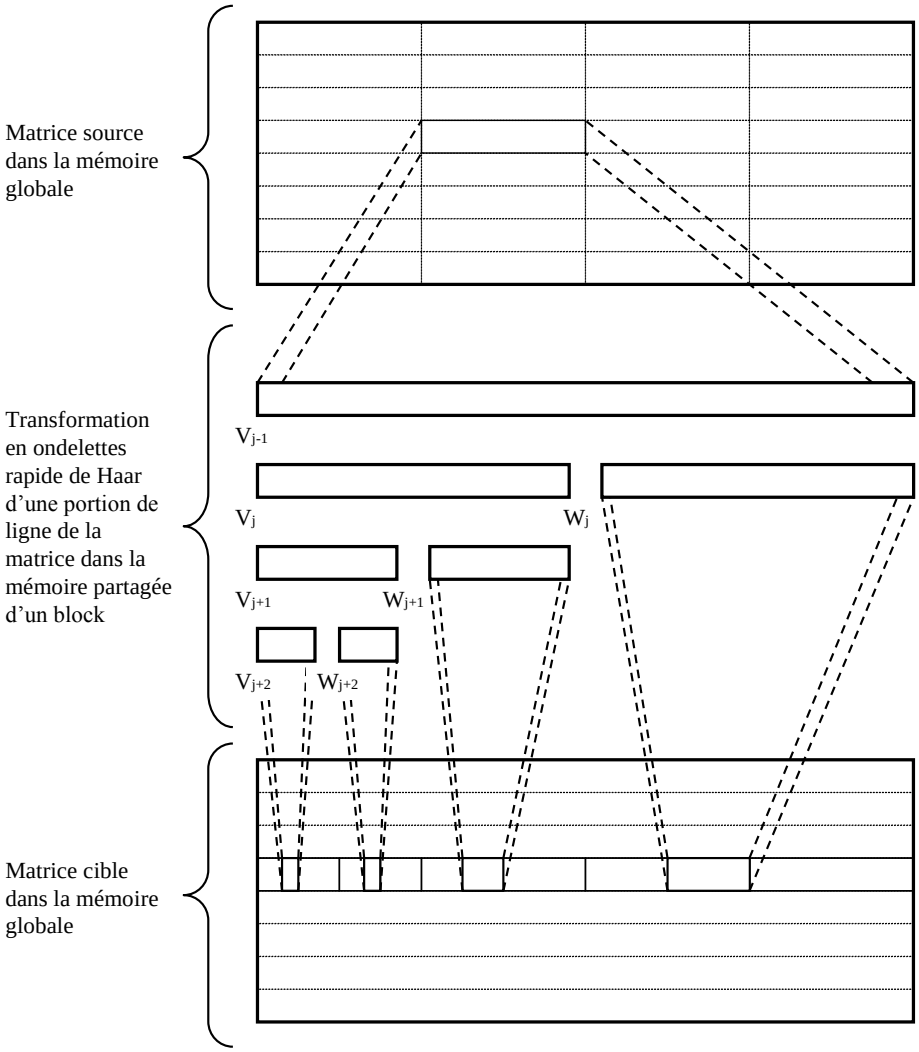


Figure 25 : Transformation en ondelettes rapide de Haar d'une matrice sur GPU.

Nous avons développé une version optimisée pour le cas où une ligne de la matrice peut être contenue entièrement dans un block. Nous travaillons alors uniquement dans la mémoire partagée et les accès à la mémoire globale en entrée et sortie sont parfaitement coalescents.

La transformation en ondelettes de Daubechies D4 se parallélise moins efficacement. La mémoire partagée ne peut pas être utilisée (sauf dans le cas où toute une ligne de la matrice peut être contenue dans un block) car les informations contenues dans les autres blocs sont nécessaires pour calculer les coefficients à chaque résolution. L'implémentation GPGPU est alors de manière générale peu performante.

Notons également que les calculs sont effectués en simple précision. Des différences importantes peuvent apparaître avec l'implémentation CPU en double précision. Cependant, compte tenu du seuillage, ces erreurs n'auront peut-être aucun effet sur la résolution du problème intégral.

5.1.2 Performances

Nous comparons sur la Figure 26 les temps de calcul de la transformée en ondelettes d'une matrice 4096x4096 entre le CPU et le GPU. Nous ne prenons pas en compte les temps de transferts de données entre l'hôte et le GPU. L'implémentation GPU de la transformée en ondelettes de Haar est très performante, nous avons une accélération d'environ 17. La transformée en ondelettes de Daubechies D4 ne se parallélise pas aussi efficacement que celle de Haar à cause de l'opérateur modulo de l'équation (12), nous ne pouvons pas utiliser la mémoire partagée pour calculer plusieurs résolutions successives. Les performances de ne sont alors pas intéressantes, l'accélération est d'environ 1,1.

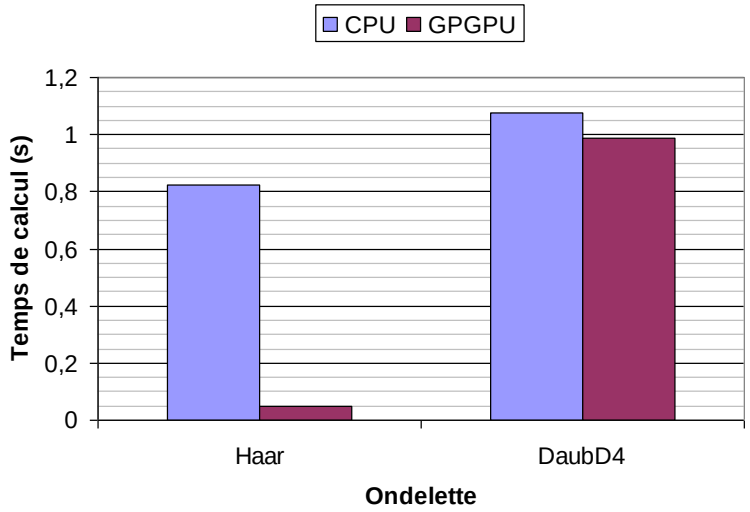


Figure 26 : Comparaisons des temps de calcul de la transformée en ondelettes d'une matrice 4096x4096 entre CPU et GPU.

Le calcul en simple précision a des conséquences sur la transformée en ondelettes. Dans le cas de l'ondelette de Haar, l'erreur relative sur les termes de la matrice entre le calcul CPU et GPU ne dépasse pas 1e-6. Par contre, dans le cas de l'ondelette de Daubechies D4, l'erreur sur certains termes dépasse les 10% voire plus. Il y a même des termes qui changent de signe entre le CPU et le GPU. Les termes sur lesquels il y a une

erreur importante sont des termes très petits relativement aux autres, l'opération de seuillage devrait les éliminer et cette perte en précision sur la transformée en ondelettes ne devrait alors pas avoir d'incidence sur la résolution du problème intégral.

5.2 Compression par ondelettes de la matrice d'interaction

5.2.1 Algorithme GPGPU

Nous utilisons le GPU pour les opérations les plus parallélisables : le calcul des blocs, leur transformation en ondelettes, le calcul de la norme (avec CuBlas) et le seuillage. Le seuillage est effectué avec un kernel CUDA dédié, nous définissons une grille de calcul en 2D et nous créons autant de threads que d'éléments de la matrice. Chaque thread seuille un seul élément de la matrice (opération (16)). Les blocs sont ensuite transférées sur l'hôte pour y être convertis en format de matrice creuse par ligne compressée. Cette opération n'est pas parallélisable. Les blocs dans leur nouveau format retournent dans la mémoire du GPU en attente de la résolution itérative. Nous pourrions également les garder dans la mémoire de l'hôte, mais compte tenu que le GPU possède 4Go de mémoire nous préférons les utiliser. Les blocs diagonaux, ainsi que les blocs jugés trop petits, sont construits sur l'hôte (en double précision). La résolution itérative est effectuée comme le montre l'algorithme présenté au paragraphe 3.9 que le bloc soit sur l'hôte où le GPU.

5.2.2 Performances

Les résolutions GPGPU donnent exactement les mêmes valeurs des capacités (à 6 chiffres significatifs) que celles obtenues sur CPU pour les deux ondelettes. Le nombre d'itérations GMRES est également identique ainsi que les taux de compression (4 chiffres significatifs). Par conséquent, la simple précision n'influence absolument pas la résolution du problème intégral. La mauvaise qualité de la transformée en ondelettes sur GPU est compensée par le seuillage.

Nous présentons les temps d'intégration et de compression matricielle entre CPU et GPGPU pour les deux ondelettes sur la Figure 27. L'ondelette la plus performante sur GPGPU est l'ondelette de Haar, elle permet une accélération de près de 17 fois (cas 3000 éléments) alors que l'accélération avec l'ondelette de Daubechies D4 ne dépasse pas 6. Nous avons vu précédemment que la transformée en cette ondelette se parallélise très mal.

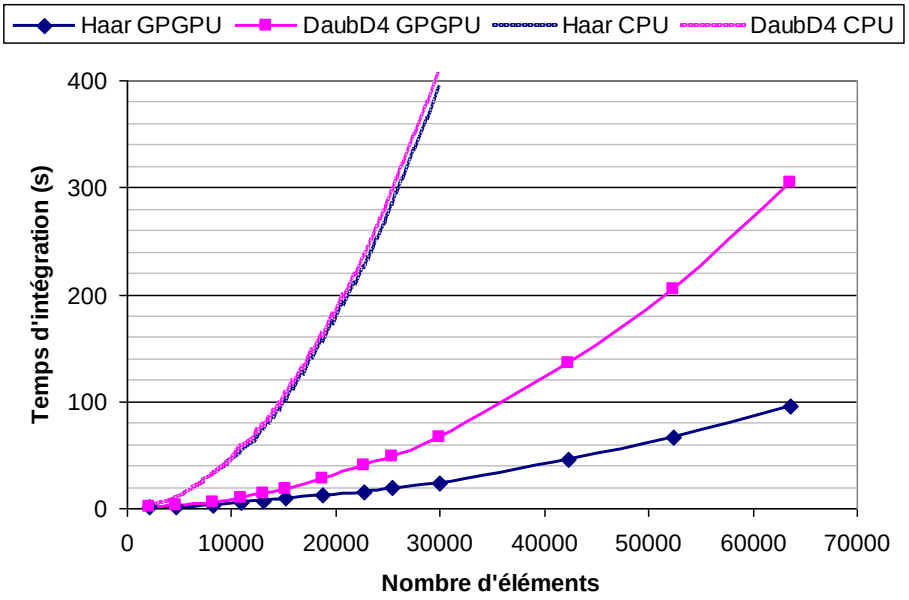


Figure 27 : Comparaison des temps d'intégration et de compression entre CPU et GPGPU.

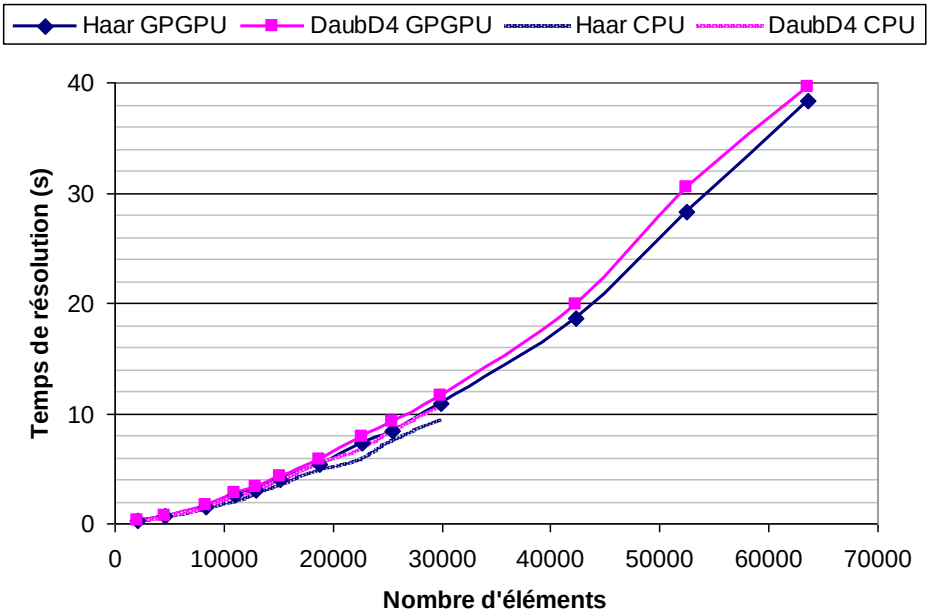


Figure 28 : Comparaison des temps de résolution entre CPU et GPGPU.

Les temps de résolution sont présentés sur la Figure 28. Les temps GPGPU sont moins bons que ceux CPU (d'environ 10-15%). L'explication provient de la grande quantité de blocs et par conséquent la grande quantité de transferts de données entre le GPU et l'hôte.

De plus les matrices étant très creuses il y a peu de calcul à effectuer, hors les GPU sont conçus pour traiter de grandes quantités de données. Stocker les blocs sur l'hôte serait plus efficace, mais la grande quantité de mémoire du GPU ne serait alors pas exploitée.

5.3 Conclusion sur la parallélisation sur architecture GPGPU

Nous venons de porter l'algorithme de compression matricielle de formulations intégrales sur architecture GPGPU. Nous obtenons une précision sur le calcul de la capacité équivalente avec l'implémentation CPU malgré un calcul en simple précision de la matrice et de sa compression. Le calcul du système d'équations compressé par l'ondelette de Haar est fortement accéléré (17x) par le processeur graphique. Par contre les gains sont seulement de 6 avec l'ondelette de Daubechies D4 car elle se parallélise moins efficacement. La résolution purement GPU n'est pas intéressante par rapport au CPU, mais nous l'utilisons tout de même afin d'exploiter la grande quantité de mémoire graphique dont nous disposons (4Go).

6 Conclusion

Nous avons dans ce chapitre appliqué des méthodes de compression matricielle par ondelettes à la matrice d'interaction afin de diminuer les besoins en mémoire vive de notre formulation intégrale. Ces méthodes ne sont pas invasives car elles ne nécessitent pas de modification des méthodes d'assemblage de la matrice d'interaction.

La matrice d'interaction ne peut pas être calculée entièrement car elle n'est pas stockable dans la mémoire vive. Elle est alors calculée par blocs. Un partitionnement de la matrice d'interaction en blocs de dimensions en puissance de deux a tout d'abord été adopté. Chaque bloc est ensuite assemblé, puis transformé en ondelettes. Cette transformation n'est pas une méthode de compression en soit, c'est le seuillage qui génère une grande quantité de zéros. Ces derniers ne sont pas stockés dans un format de matrice creuse, d'où la réduction des besoins en mémoire vive. Nous avons développé un critère de seuillage basé sur la dégradation relative de la norme. Nous avons implémenté deux ondelettes, l'ondelette de Haar et l'ondelette de Daubechies D4 qui sont toutes les deux des ondelettes orthonormales à support compact. Nous avons obtenu des taux de compression

de l'ordre de 97% (condensateur plan de 30.000 éléments pour une dégradation relative de la norme de 1e-4).

L'inconvénient des méthodes de compression matricielle par ondelettes est qu'elles ne réduisent pas la complexité de la construction du système d'équations qui est toujours parabolique. Nous avons alors utilisé la parallélisation massive sur GPGPU afin d'accélérer les calculs. Nous avons obtenu des accélérations de l'ordre de 17 (30.000 degrés de liberté) avec l'ondelette de Haar et 6 avec l'ondelette de Daubechies, cette dernière se parallélisant moins efficacement.

Nous avons ensuite tenté un couplage avec les matrices hiérarchiques (voir l'Annexe D). Cette méthode permet via une analyse géométrique de partitionner la matrice d'interaction en blocs très homogènes et par conséquent hautement compressibles à priori. Les résultats préliminaires montrent que l'association des matrices hiérarchiques à la compression par ondelettes se révèle prometteur d'un point de vue taux de compression pour une erreur donnée. Cependant la très grande quantité de blocs nuit à la parallélisation sur GPGPU qui ne se révèle pas performante. De plus, les blocs générés sont de tailles quelconques et il est nécessaire de les agrandir ce qui introduit une difficulté supplémentaire.

Nous proposons dans le chapitre suivant d'appliquer la compression matricielle par ondelettes dans le cadre d'un calcul de capacités parasites sur un dispositif de complexité industrielle.

7 Références

[Ajdari 10] J. Ajdari, F. Hoxha, « Parallel implementation of 2D Daubechies - D4 transform in a cluster, » Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on , vol., no., pp.377-382, 2010.

[Ardon 10] V. Ardon, « Méthodes numériques et outils logiciels pour la prise en compte des effets capacitifs dans la modélisation CEM de dispositifs d'électronique de puissance », Thèse de doctorat, INPG, Grenoble, France, 2010.

[Buchau 03] A. Buchau, W.M. Rucker, O. Rain, V. Rischmuller, S. Kurz, S. Rjasanow, « Comparison between different approaches for fast and efficient 3-D BEM computations, » Magnetics, IEEE Transactions on , vol.39, no.3, pp. 1107- 1110, May 2003.

[Bucher 03] H. Bucher, L. Wrobel, W. Mansur, C. Magluta, « Fast solution of problems with multiple load cases by using wavelet-compressed boundary element matrices », Communications in Numerical Methods in Engineering, 19: 387-399, 2003.

[Bucher 04] H. Bucher, L. Wrobel, W. Mansur, C. Magluta, « On the block wavelet transform applied to the boundary element method », Engineering analysis with boundary elements, vol. 28, no. 6, pp. 571-581, 2004.

[Ebrahimnejada 10] L. Ebrahimnejada, R. Attarnejad, « Fast solution of BEM systems for elasticity problems using wavelet transforms », International Journal of Computer Mathematics, vol. 87, no. 1, pp. 77-93, 2010.

[Greengard 87] L. Greengard and V. Rokhlin, « A fast algorithm for particle simulations, Journal of Computational Physics », vol. 73, Issue 2, pp. 325-348, Dec 1987.

[Greengard 99] H. Cheng, L. Greengard, and V. Rokhlin, « A Fast Adaptive Multipole Algorithm in Three Dimensions », Journal of Computational Physics 155, 468–498, 1999.

[Jianjun 11] W. Jianjun; L. Wenlong; L. Xiaojun; J. Xiangqian, « High-speed parallel wavelet algorithm based on CUDA and its application in three-dimensional surface texture analysis », Electric Information and Control Engineering (ICEICE), 2011 International Conference on , vol., no., pp.2249-2252, 15-17 April 2011.

[Koro 03] K. Koro, K. Abe, « A pratical determination strategy of optimal threshold parameter for matrix compression in wavelet BEM », International Journal for Numerical Methods in Engineering, vol. 57, no. 2, pp. 169-191, 2003.

[Mikulic 04] E. Mikulic, « Haar Wavelet Transform », http://dmr.ath.cx/gfx/haar/, 2004.

[Rjasanow 07] S. Rjasanow, O. Steinbach, « The Fast Solution of Boundary Integral Equations », Springer, 2007.

[Rubeck 11] C. Rubeck, B. Bannwarth, O. Chadebec, B. Delinchant, J-P. Yonnet, and J-L. Coulomb, « JCuda vectorized and parallelized computation strategy for solving integral equations in electromagnetism on a standard personal computer », COMPUMAG 2011, Australie, 2011.

[Saad 00] Y. Saad, « Iterative Methods for Sparse Linear Systems - Second Edition », SIAM, 2000.

[Sadiku 05] M.N.O. Sadiku, C.M. Akujuobi, R.C. Garcia, « An introduction to wavelets in electromagnetics », IEEE Microwave Magazine, pp. 63 - 72 vol. 6, no. 2, June 2005.

[Scheiblich 09] C. Scheiblich, V. Kolitsas and W. M. Rucker, « Compression of the Radiative Heat Transfer BEM Matrix of an Inductive Heating System Using a Block-Oriented Wavelet Transform », IEEE Trans. Magn., vol. 47, no. 3, pp. 1712-1715, Mar. 2009.

[Scheiblich 11] C. Scheiblich, R. Banucu, V. Reinauer, J. Albert and W. M. Rucker, « Parallel Hierarchical Block Wavelet Compression for an Optimal Compression for 3-D BEM Problems », IEEE Trans. Magn., vol. 47, no. 5, pp. 1386-1389, May 2011.

[Stollnitz 95] E. Stollnitz, A. DeRose, D. Salesin, « Wavelets for computer graphics: a primer.1 », Computer Graphics and Applications, IEEE , vol.15, no.3, pp.76-84, May 1995.

Chapitre V

Application : Calcul des capacités parasites d'un variateur de vitesse pour la CEM

Sommaire

Résumé

Nous proposons dans ce chapitre d'évaluer les performances de la compression par ondelettes sur GPGPU. Nous appliquons notre méthodologie sur un dispositif de complexité industrielle : le PCB d'un variateur de vitesse.

1 Introduction

Nous présentons un calcul de capacités parasites sur un variateur de vitesse : l'ATV71. Il s'agit d'un dispositif du génie électrique de complexité industrielle. Il est constitué d'un nombre important de conducteurs disposés en couche minces. La distance entre les conducteurs maillés est plus fine que la taille des éléments, ce qui représente une difficulté majeure en terme de modélisation. Ce dispositif serait très difficile à modéliser en éléments finis par exemple. Avec la formulation intégrale développée au chapitre II, seul les surfaces des conducteurs sont maillées, ce qui réduit considérablement le nombre de degrés de liberté nécessaire à la modélisation. Cependant cette formulation avec les 80.000 éléments de maillage génèrerait une matrice de 50Go environ sans méthode de compression. Nous proposons d'appliquer notre méthodologie de compression par ondelettes sur ce dispositif.

Nous commençons le chapitre par une introduction à la compatibilité électromagnétique, en particulier la notion de capacité parasite puis nous présentons le variateur de vitesse. Nous comparons ensuite les temps de calcul entre CPU et GPGPU et faisons quelques remarques sur la qualité des solutions obtenues.

2 Notion de capacités parasites

Les systèmes électromagnétiques peuvent être perturbés par leur propre fonctionnement et/ou l'environnement électromagnétique dans lequel ils se trouvent. La compatibilité électromagnétique (CEM) a pour but de garantir le bon fonctionnement des appareils c'est-à-dire qu'ils ne se perturbent pas eux même, qu'ils ne perturbent pas leur environnement et enfin qu'ils ne soient pas perturbés par leur environnement. Nous distinguons deux catégories de perturbations électromagnétiques : les perturbations conduites et les perturbations rayonnées. Dans le premier cas les perturbations sont transmises via des conducteurs (câbles, plan de masse), voire même des diélectriques générateurs de capacités parasites, c'est ce dernier cas qui fera l'objet de notre étude. Les perturbations rayonnées se transmettent à travers des ondes électromagnétiques principalement dans l'air.

Pour comprendre la notion de capacités parasites, regardons la Figure 1. Elle représente un circuit électronique composé par deux pistes de cuivre respectivement aux potentiels *V* et *V'* [V] séparées par un substrat diélectrique de permittivité relative ε_r. L'interconnexion entre ces deux pistes de conducteur peut être représentée par une capacité. Le courant électrique à travers un condensateur est donné par :

$$I = \omega C(V - V') \tag{1}$$

Où *I* est le courant électrique [A], ω la fréquence ou pulsation [rad/s], et *C* la capacité [F]. Le courant est nul dans le cas d'une tension continue, le diélectrique se comporte comme un isolant. Par contre en fréquentiel, le diélectrique devient perméable au courant électrique proportionnellement à la fréquence. Cette rupture partielle de l'isolation entre les pistes est à l'origine des perturbations capacitives.

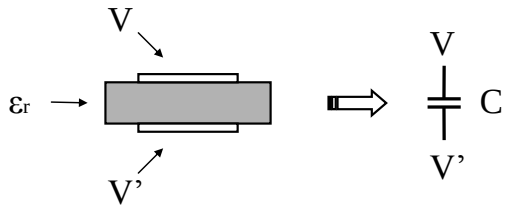


Figure 1 : Origine des capacités parasites entre deux pistes d'un circuit électronique.

La modélisation des perturbations en mode commun nécessite le calcul des capacités entre tous les conducteurs [Aimé 09]. Nous rappelons que ces capacités C_{ij} entre les conducteurs *i* et *j* peuvent être obtenues depuis les distributions de charges à la surface des conducteurs par :

$$C_{ij} = \sum_{k=1}^{N(j)} \varepsilon_{rk}^{(j)} \, q_k^{(j)} \tag{2}$$

Où *N(j)* est le nombre d'élément de la région *j*, q_k est la charge de l'élément *k* [C] et ε_{rk} est la permittivité diélectrique relative du milieu environnant à l'élément *k*. Le potentiel V_i de la région *i* est choisi à 1V et tous les autres conducteurs sont soumis à 0V. Ces capacités sont appelées capacité de Maxwell, les capacités utilisées dans les circuits électroniques sont les capacités de Kirchhoff C'_{ii} données par :

$$\begin{cases} C'_{ii} = \sum_{j=1}^{n} C_{ij}, & \text{pour tout i} \\ C'_{ij} = -C_{ij}, & \text{si i} \neq \text{j} \end{cases} \quad (3)$$

L'ensemble des capacités entre les conducteurs forme la matrice des capacités, elle peut être utilisée dans un solveur circuit pour modéliser les perturbations en mode commun. Nous proposons de calculer la matrice des capacités avec les modèles numériques développés précédemment sur un exemple de complexité industrielle : un variateur de vitesse ATV71.

3 Présentation du variateur de vitesse ATV71

Nous présentons le variateur de vitesse ATV71 sur la Figure 2. C'est un convertisseur de fréquences pour moteurs synchrones et asynchrones triphasés dans la gamme 230V/5,5kW. Il est commercialisé par « Schneider and Toshiba Inverter Europe » (STIE). Les domaines d'applications sont les machines tournantes, les ascenseurs, etc.

Figure 2 : Variateur de vitesse ATV71.

La modélisation CEM capacitive est complexe : les circuits électroniques sont composés de couches minces de conducteurs en vis-à-vis (4 couches pour certaines régions) difficiles à modéliser (Figure 3). Des perturbations peuvent également émaner de l'armature qui sert également de plan de masse. Le maillage capacitif est composé de 27 régions maillées en 80144 triangles. Par conséquent 27 résolutions seront nécessaires pour calculer la matrice des capacités.

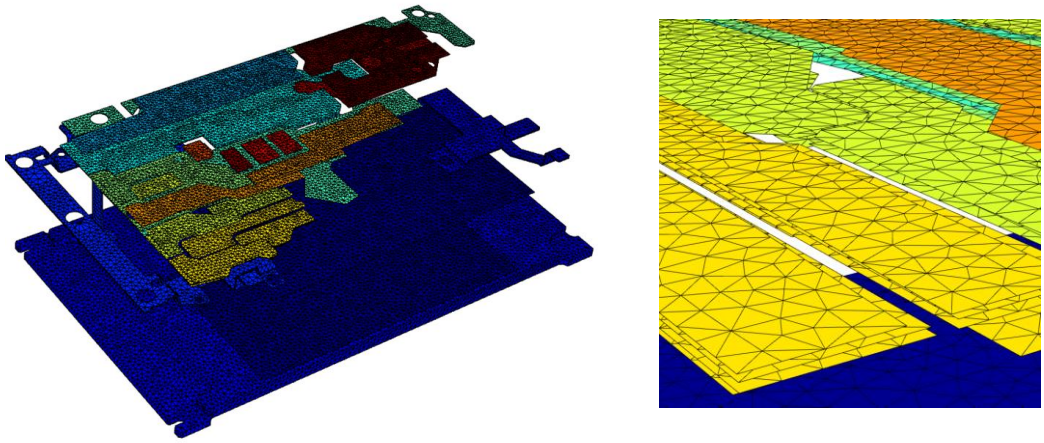


Figure 3 : Maillage capacitif du variateur de vitesse.

4 Performances du calcul des distributions de charges

4.1 Méthodologie et configuration matérielle

Nous utilisons pour le calcul des distributions de charges la formulation électrostatique en collocation à l'ordre 0 développée dans le chapitre II (3.2.3).

L'algorithme de compression matricielle par ondelettes est utilisé sur CPU et GPGPU avec des blocs de tailles maximales respectivement de 512x512 et 4096x4096. Les blocs diagonaux sont de tailles maximales 256x256. L'ondelette utilisée est l'ondelette de Haar. Le niveau de l'octree est fixé à 7.

Pour améliorer la convergence de la résolution itérative nous utilisons un préconditionneur par blocs inversés construit à partir d'une décomposition LU des blocs diagonaux.

La configuration matérielle est la même qu'aux chapitres précédents, c'est-à-dire que le CPU est un Intel Xeon cadencé à 2.67 GHz en monocoeur et la carte graphique utilisée est une Tesla C1060 (240 cœurs, 4Go de mémoire). Les calculs sont effectués en simple précision sur le GPU et en double précision sur le CPU.

4.2 Temps de calcul

Nous comparons sur la Figure 4 les temps de calcul des deux architectures (CPU et GPGPU) pour différents paramètres de seuillage et nous indiquons les taux de compression qui en découlent. Nous voyons que les temps d'intégration sont identiques quelque soit le seuillage car l'ensemble de la matrice d'interaction est calculé. Les temps de résolution sont différents car en format matrice creuse la complexité d'un produit matrice vecteur dépend du nombre d'éléments stockés. Nous remarquons que les temps de résolution GPGPU sont quasiment identiques dans les deux cas, cela signifie que les opérations dominantes sont les transferts de données entre le GPU et l'hôte. Nous obtenons une accélération entre CPU et GPGPU de 16,6 pour l'intégration et 4,6 et 2,6 pour les résolutions correspondantes respectivement aux taux de compression de 97% et 98%. Les accélérations totales sont de 6,2 et 4,4 pour respectivement 97% et 98% de compression.

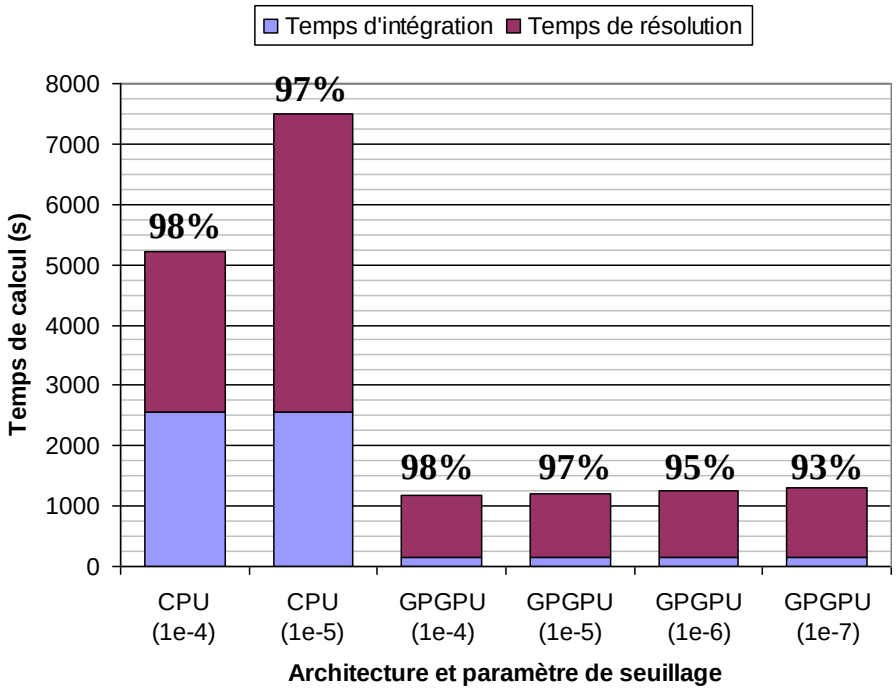


Figure 4 : Temps de calcul pour les architectures CPU et GPGPU. Nous indiquons en gras les taux de compression obtenus.

Evidemment, aucune comparaison n'est effectuée avec une approche en matrice non compressée puisque la résolution de notre problème avec ce type de technique n'est pas envisageable. Nous présentons sur la Figure 5 le taux de compression et l'occupation

mémoire en fonction du paramètre de seuillage. Nous avons pu atteindre un taux de compression plus faible sur GPGPU car nous bénéficions des 4Go de mémoire graphique.

Epsilon	Taux de compression	Occupation mémoire (Mo)
1e-4	98,09	936
1e-5	97,07	1436
1e-6	95,44	2235
1e-7	92,84	3509

Figure 5 : Taux de compression et occupation mémoire en fonction du paramètre de seuillage.

4.3 Qualité de la résolution

Nous nous intéressons à la région numéro 12, la résolution associée est celle qui converge le plus difficilement des 27. Nous présentons les résultats associés à la résolution la plus précise, le critère de dégradation de la norme est 1e-7. La Figure 6 montre la distribution de charges, nous pouvons remarquer un bruit numérique avec la compression par ondelettes.

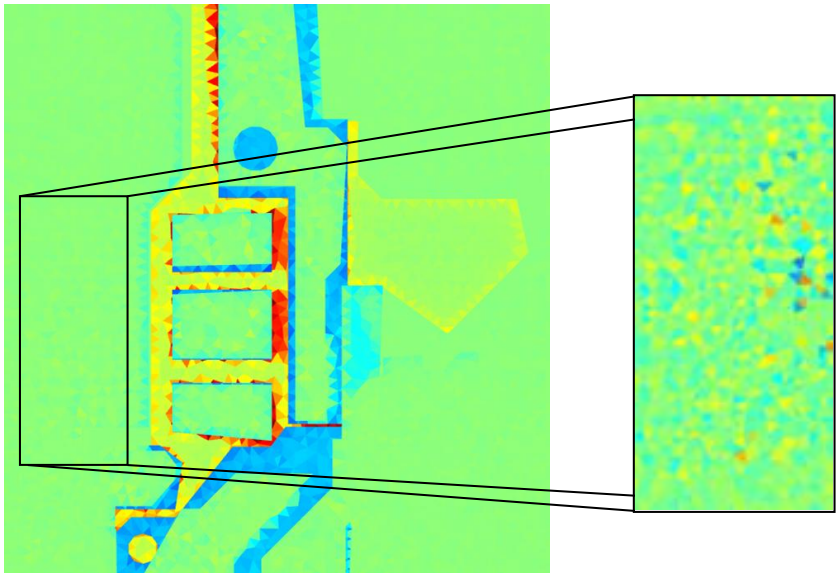


Figure 6 : Distribution de charges sur le variateur de vitesse. Du bruit apparaît lors de la compression par ondelettes.

5 Calcul des matrices de capacités

Nous ne disposons pas des informations sur les éléments du maillage en contact avec des matériaux diélectriques. Nous ne les considèrerons donc pas et tacherons simplement

de retrouver les ordres de grandeurs caractéristiques des capacités qui sont, pour les plus importantes, de l'ordre de grandeur du pF [Aimé 09].

Nous présentons à la Figure 7 des extraits de la matrice des capacités obtenue avec la compression par ondelettes. De manière générale la matrice est proche de la symétrie, elle ne l'est pas exactement à cause de la technique d'intégration qui est imparfaite et l'intégration par la méthode de collocation qui ne produit pas une matrice d'interaction symétrique. Cependant nous notons la présence d'un certain nombre de capacités négatives dans la matrice. Ces quantités ne sont bien sûr pas physiques et proviennent des erreurs introduites par la formulation et surtout la compression matricielle.

Région	1	2	3	4	5	6
1	8,48E-13	2,02E-12	1,29E-12	7,61E-12	2,98E-12	2,73E-12
2	2,03E-12	2,67E-13	8,08E-11	3,24E-13	3,22E-13	1,25E-13
3	1,28E-12	8,08E-11	6,10E-12	3,94E-12	8,52E-13	3,06E-12
4	7,61E-12	3,08E-13	3,96E-12	5,92E-13	1,29E-15	2,44E-14
5	2,97E-12	3,30E-13	8,43E-13	4,80E-15	1,33E-14	2,02E-14
6	2,73E-12	1,67E-13	3,02E-12	2,54E-14	1,58E-14	1,22E-12
7	6,09E-13	1,29E-14	6,41E-14	1,59E-13	4,67E-16	6,05E-16
8	8,92E-13	2,44E-14	8,85E-14	5,29E-15	1,93E-14	5,13E-17
9	7,65E-14	8,14E-14	4,59E-13	3,25E-14	5,33E-14	5,98E-13
10	6,22E-14	6,28E-14	1,20E-13	1,82E-14	4,16E-14	8,39E-13
11	2,74E-13	7,29E-14	1,08E-13	1,98E-14	2,08E-13	1,04E-12
12	7,73E-14	-1,80E-13	7,69E-13	1,48E-13	-6,92E-15	1,31E-12
13	1,67E-12	-1,27E-13	2,59E-13	7,88E-14	2,13E-13	4,44E-13
14	6,74E-13	-3,97E-14	1,76E-13	4,03E-16	-1,31E-15	4,32E-13
15	2,94E-14	2,21E-14	5,77E-16	8,04E-16	8,96E-16	4,13E-14
16	4,41E-13	1,70E-14	6,96E-14	4,13E-15	4,25E-16	3,32E-13
17	3,30E-13	8,10E-15	2,73E-14	2,91E-15	2,05E-16	2,75E-13
18	5,68E-13	2,56E-14	7,89E-14	9,20E-15	3,65E-16	1,40E-12
19	6,87E-13	1,50E-13	-1,09E-13	7,11E-14	1,67E-13	1,79E-14
20	5,68E-14	2,46E-14	5,79E-15	2,87E-15	2,31E-16	8,44E-14
21	2,96E-14	-1,76E-14	2,58E-14	-2,91E-16	-7,20E-17	6,11E-13
22	7,21E-14	1,37E-14	-9,17E-15	1,87E-15	9,20E-14	2,22E-15
23	1,40E-13	1,93E-14	-1,23E-14	2,12E-15	1,87E-14	5,01E-15
24	1,14E-13	1,16E-14	-5,79E-15	8,83E-16	1,96E-15	8,48E-15
25	2,23E-14	1,67E-14	6,85E-14	1,29E-13	6,53E-15	4,22E-15
26	3,37E-14	6,51E-14	3,02E-13	1,78E-13	5,62E-15	1,80E-14
27	1,26E-13	-1,16E-15	1,31E-13	2,97E-13	3,29E-16	1,03E-14

Figure 7 : Extrait de la matrice des capacités obtenue avec la compression par ondelettes sur GPGPU (tolérance 1e-7).

Nous tentons ici de caractériser la qualité de la matrice des capacités en fonction de sa symétrie. Nous proposons le critère suivant :

$$\alpha = \frac{norme\,(C - C^T)}{norme\,(C)} \cdot 100 \qquad (4)$$

Où la norme est prise au sens de Frobenius. Moins la matrice sera symétrique et plus le critère α sera élevé. Nous obtenons les résultats présentés sur la Figure 8. Nous observons que plus le système d'équations est compressé et moins bonne est la qualité de la matrice des capacités, ce résultat est tout à fait cohérent.

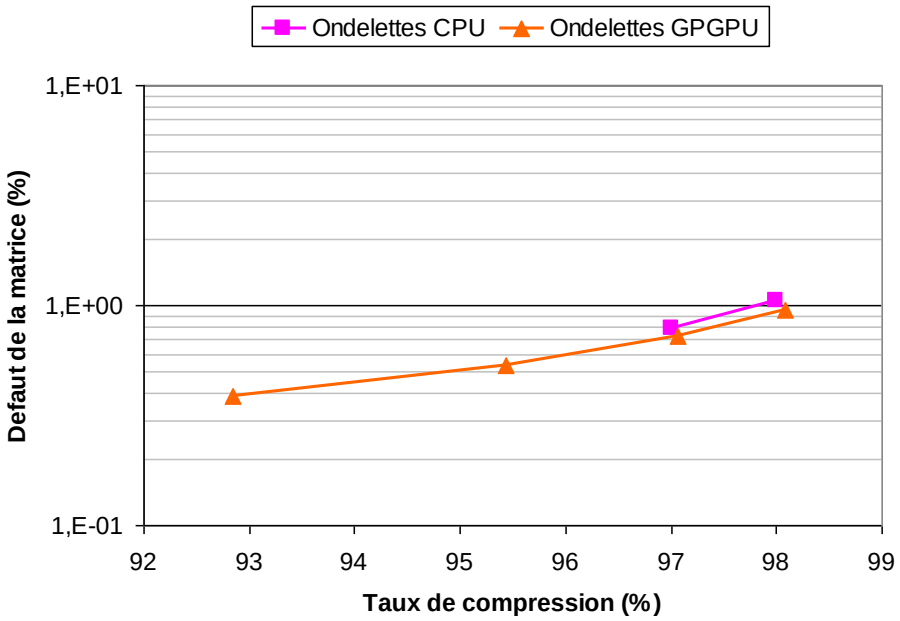


Figure 8 : Défaut de symétrie dans la matrice des capacités.

6 Conclusion

Nous avons appliqué notre méthodologie de compression matricielle par ondelettes sur GPGPU dans le cadre d'un calcul de capacités parasites sur un variateur de vitesse. Nous retrouvons les ordres de grandeurs caractéristiques des capacités parasites, avec toutefois quelques erreurs telles que des capacités négatives. Ces erreurs peuvent avoir pour origine du bruit numérique lors des produits matrice-vecteur, en effet le second membre du système d'équations subit une transformation en ondelettes et le vecteur solution une transformation inverse. Une autre origine d'erreurs peut être les interactions proches qui ne

sont pas toutes dans les blocs diagonaux et qui par conséquent sont dégradés par la compression, d'où notre intérêt de coupler notre méthodologie avec les matrices hiérarchiques. Nous pourrions également penser au calcul simple précision, cependant la modélisation CPU est effectuée en double précision et nous retrouvons les mêmes erreurs.

Nous avons appliqué avec succès la compression matricielle par ondelettes sur un dispositif industriel. Cette méthode de compression est peu invasive, c'est-à-dire qu'elle est aisée à mettre en œuvre. Elle est également rapide grâce au parallélisme sur GPGPU. De plus, la connaissance explicite des blocs diagonaux permet de réaliser des préconditionneurs efficaces.

7 Références

[Aimé 09] J. Aimé, « Rayonnement des convertisseurs statiques. Application à la variation de vitesse », Thèse de doctorat, INPG, Grenoble, France, 2009.

[Ardon 10] V. Ardon, « Méthodes numériques et outils logiciels pour la prise en compte des effets capacitifs dans la modélisation CEM de dispositifs d'électronique de puissance », Thèse de doctorat, INPG, Grenoble, France, 2010.

Conclusion générale et perspectives

L'objectif principal de ces travaux a été d'améliorer les performances (temps de calcul et précision) des méthodes intégrales. Ces méthodes sont bien adaptées à la modélisation des systèmes électromagnétiques notamment parce qu'elles ne nécessitent pas le maillage des matériaux inactifs tel que l'air. Cependant elles souffrent de certaines lacunes, dont trois principales que nous avons identifiées et autour desquelles se sont articulés nos travaux de thèse.

Le premier axe d'étude a été d'hybrider le calcul numérique d'intégrales, apprécié pour sa vitesse mais qui peut être imprécis, au calcul analytique, qui lui est apprécié pour sa grande précision. Nous avons développé une méthode analytique de calcul du potentiel et du champ électrostatique créés par des polygones uniformément chargés. Notons que cette méthode s'applique également au calcul du potentiel et du champ magnétostatiques créés par des aimants par exemple.

Le deuxième axe a été d'accélérer les calculs de la matrice d'interaction et de sa résolution itérative qui sont tous deux de complexité parabolique. Nous avons pour ce faire utilisé plusieurs processeurs, c'est le parallélisme. Nous avons testé le multicoeur, le clustering à travers un petit réseau, et l'architecture GPGPU. Cette architecture particulière a été l'objet central de notre étude sur les apports du parallélisme. Pour chacune de ces architectures parallèles, des stratégies différentes de partitionnement, de communication et de synchronisation des tâches ont été mises en place.

Le troisième et dernier axe a été de réduire les besoins paraboliques en mémoire vive des méthodes intégrales. Ce dernier point est souvent le facteur limitant l'attrait de ces méthodes. Nous avons compressé la matrice d'interaction par ondelettes. C'est une méthode qui a l'avantage d'être peu invasive mais elle nécessite le calcul de l'ensemble de

la matrice d'interaction ce qui est très couteux. La méthode a alors été portée sur l'architecture GPGPU. La compression par ondelettes est une compression à perte, la difficulté est de contrôler les erreurs introduites par la compression. Nous avons développé un critère de compression basé sur la dégradation relative de la norme de Frobenius des blocs matriciels.

Finalement nous avons appliqué notre méthodologie sur un dispositif du génie électrique de complexité industrielle : le variateur de vitesse ATV71. Nous avons calculé les capacités parasites entre toutes les pistes de cuivre. Nous retrouvons les ordres de grandeurs des capacités parasites. Quelques erreurs subsistent toutefois à cause de la compression.

Nous avons tout au long de cette étude illustré les méthodologies sur un exemple simple d'une formulation intégrale en potentiel électrostatique résolue numériquement par la méthode de collocation à l'ordre zéro. Ces méthodologies ne sont à priori pas limitées à cette formulation et devraient être pour la plupart applicables à d'autres. Le point le plus sensible est le calcul en simple précision qui pourrait se montrer insuffisant pour certaines formulations.

Les pistes que nous pourrions explorer dans la suite de ces travaux seraient par exemple de projeter les équations intégrales par la méthode de Galerkin, toujours à l'ordre zéro. La difficulté réside dans la double intégration qui complexifie le calcul des interactions et risque de réduire la coalescence des accès mémoires. Notons que le débat est toujours ouvert dans notre communauté sur l'intérêt d'augmenter le nombre d'éléments (c'est-à-dire de raffiner le maillage) plutôt que l'ordre des éléments. Dans le contexte du calcul parallèle la solution la plus adaptée est l'augmentation du nombre d'éléments d'ordre 0 accompagnée d'une méthode de compression matricielle performante et simple.

Nous pourrions également continuer d'explorer la piste du couplage entre la compression par ondelettes et les matrices hiérarchiques. Les résultats préliminaires sont encourageants mais nous aurions besoin de développer un critère de compression plus robuste qui tiendrait compte de l'agrandissement artificiel des blocs matriciels.

Enfin, un dernier axe intéressant serait de développer le clustering. Toutes les méthodes de construction de la matrice d'interaction par blocs s'y prêtent bien à priori. En effet les

assemblages des blocs matriciels sont indépendants les uns des autres, il est donc relativement aisé de les distribuer. Il est bien sûr nécessaire d'utiliser une infrastructure réseau adaptée, nous avons vu que le coût des communications était le facteur limitant les performances.

Annexe A

Compléments à l'introduction au calcul hautes performances

Sommaire

1 Optimisation des codes de calcul

Nous présentons ici quelques techniques d'optimisation des codes de calcul numérique. Il est en effet important, une fois un code écrit et validé, d'optimiser sa vitesse d'exécution [Dowd 98]. Pour se faire, nous tiendrons essentiellement compte du fonctionnement de l'ordinateur.

1.1 Division et multiplication d'un entier par deux

Le coût d'une division est très important, c'est pourquoi il faut utiliser cet opérateur le moins possible. Il existe par exemple une astuce pour diviser un entier par 2, il suffit de se décaler d'un bit sur la droite. Cela s'écrit n>>=1 au lieu de n/=2, le coût n'est plus que d'un unique cycle d'horloge. De la même manière un entier se multiplie facilement par deux via un déplacement d'un bit sur la gauche.

1.2 Appels à des sous fonctions

Un programmeur a souvent tendance, par soucis de clarté dans la lecture de son code, à faire appel à des sous fonctions. Soit l'exemple suivant :

```
for (i = 0; i < n; i++) {
    A[i] = A[i] + B[i] * C;
}
```

Qui peut également s'écrire :

```
for (i = 0; i < n; i++) {
    A[i] = mAdd(A[i], B[i], C)
}
```

Avec la sous fonction associée :

```
double mAdd(A, B, C) {
    return A + B * C;
}
```

Cet exemple est particulièrement flagrant de la perte du pipelining. Il en résulte un ralentissement de 50 à 100 fois ! En C/C++ il est possible de déclarer ce genre de sous fonctions dans le préprocesseur ou en inlining[1], mais en Java ces mécanismes n'existent pas.

1.3 Boucles avec conditions

1.3.1 Condition invariante

Soit la boucle suivante qui effectue un test à chaque itération :

```
for (i = 0 ; i < n ; i++) {
    if (alpha < 0.0) {
        A[i] = A[i] + B[i] * C
    } else {
        A[i] = 0.0
    }
}
```

Le test donnera toujours le même résultat au sein de cette boucle, il peut alors être sorti de la boucle :

```
if (alpha < 0.0) {
    for (i = 0 ; i < n ; i++) {
        A[i] = A[i] + B[i] * C
    }
} else {
    for (i=0 ; i<n ; i++) {
        A[i] = 0.0
    }
}
```

Dans la nouvelle version du programme, n-1 copies du test ont été supprimées !

[1] Mécanisme en C/C++ qui permet au compilateur de recopier une fonction là où elle est appelée. L'exécutable est plus volumineux mais plus rapide car il n'y a plus d'appel de fonctions.

1.3.2 Condition dépendant de l'index

Soit un test qui est vrai pour un certain nombre continu d'index i, puis faux le restant de la boucle :

```
for (i=0 ; i<n ; i++) {
    if (i < k) {
        A[i] = A[i] + B[i] * C
    } else {
        A[i] = 0.0
    }
}
```

L'algorithme peut s'écrire en deux boucles :

```
for (i=0 ; i<k ; i++) {
    A[i] = A[i] + B[i] * C
}
for (i=k ; i<n ; i++) {
    A[i] = 0.0
}
```

Tous les tests ont été éliminés, de plus le pipelining est optimal. Cette optimisation peut par exemple facilement s'appliquer sur l'algorithme de Gauss-Seidel.

1.4 Division par une constante

Soit la boucle suivante :

```
for (i=0 ; i<n ; i++) {
    A[i] = A[i] / sqrt(x*x+y*y)
}
```

Cette boucle contient une division par une constante, hors les divisions sont très coûteuses. De plus, le terme au dénominateur est également coûteux à calculer, il doit être calculé hors de la boucle :

```
cste = 1 / sqrt(x*x+y*y)
for (i=0 ; i<n ; i++) {
    A[i] = A[i] * cste
}
```

Le code a été accéléré d'un facteur de 8 à 10 uniquement grâce à la suppression de la division. L'accélération est plus importante encore due au fait que la fonction racine carrée n'est plus appelée à chaque itération. Cet exemple particulièrement pathologique montre qu'il n'est pas nécessaire de complexifier l'algorithme pour être plus performant.

1.5 Boucles imbriquées

Soit l'algorithme très connu de multiplication matricielle. Classiquement il s'écrit (pour une matrice n x n) :

```
for (i=0 ; i<n ; i++) {
    for (j=0 ; j<n ; j++) {
        sum = 0
        for (k=0 ; k<n ; k++)
            sum = sum + A[i,k] * B[k,j]
        }
        C[i,j] = sum
    }
}
```

Ici, l'accès à B[k,j] n'est pas coalescent ! L'algorithme peut être réécrit de cette façon :

```
for (i=0 ; i<n ; i++) {
    for (j=0 ; j<n ; j++) {
        C[i,j] = 0
    }
}

for (k=0 ; k<n ; k++) {
    for (i=0 ; i<n ; i++) {
        scale = A[i,k]
```

```
            for (j=0 ; j<n ; j++)
                C[i,j] = C[i,j] + B[k,j] * scale
            }
        }
    }
```

L'accès coalescent à B[k,j] est retrouvé pour le prix d'une variable supplémentaire dans l'algorithme. En réalité ce prix est nul car si le compilateur fait bien son travail elle sera placée en cache au plus près du processeur.

2 Introduction à la programmation CUDA

2.1 Premier programme CUDA

Soit un programme qui pour un tableau X[N] de taille N effectue l'opération Y[i] = f(X[i]). Définissons une grille de calcul de N threads. Chaque thread effectue alors une seule opération. Le kernel correspondant s'écrit :

```
__global__ void kernel_fx(float* X, float* Y,…)
    {…
    float x = X[threadID] ;
    float y = f(x) ;
    Y[threadID] = y ;
    …}
```

La numérotation des threads correspond ici aux indices des éléments dans les vecteurs X et Y. Les variables x et y sont locales à chaque thread, elles sont stockées dans les registres de chaque cœur. Le mot clef __global__ indique qu'il s'agit d'un kernel.

La sous fonction f(x) s'écrit simplement :

```
__device__ float f(float x)
{
    float y = 2 * x + 5 ;
    return y ;
}
```

Le mot clef __device__ indique que la fonction s'exécute sur la carte graphique. Il est également possible d'utiliser les mécanismes d'inlining du C++.

La topologie choisie construit des blocks 1D de 512 threads chacun. Maintenant que les blocks sont définis, il faut définir le nombre de blocks sur la grille. La grille de calcul est définie en 1D également, ce qui est bien adapté à une opération vectorielle. La grille de calcul est construite au lancement du kernel. Les tableaux contenus dans la mémoire globale de la carte graphique sont donnés en argument, ainsi que divers paramètres (ici la taille du tableau). Le programme principal s'écrit alors :

```
#define BLOCK_SIZE 512
…
int main(int argc, char *argv[])
    {…
    dim3 dimBlock(BLOCK_SIZE,1,1) ;
    dim3 dimGrid((N-1)/BLOCK_SIZE+1,1) ;
    kernel_fx <<< dimGrid, dimBlock >>> (X, Y, N) ;
    …}
```

Le nombre de threads est supérieur à N, en fait il est du multiple de BLOCK_SIZE supérieur. Un test pour éviter les débordements mémoires sera mis en place dans le kernel CUDA. La grille étant complètement définie, le kernel doit pouvoir se positionner dessus. Le paramètre threaID est maintenant déterminé, le kernel s'écrit :

```
__global__ void kernel_fx(float* X, float* Y, int N)
{
    int threadID = blockIdx.x* blockDim.x+ threadIdx.x ;
    if (threadID < N) // Vérification pour ne pas déborder du tableau
    {
        float x = X[threadID] ;
        float y = f(x) ;
        Y[threadID] = y ;
    }
}
```

Les blocks ont des coordonnées et une taille suivant chaque dimension. Dans chacun de ses blocks, les threads sont également identifiés par leurs coordonnées suivant chaque dimension. Ici seul les coordonnées sur x apparaissent car on est en 1D. Les accès à la mémoire globale sont coalescents. En effet, chaque threadID+1 accède au block mémoire suivant de celui appelé par threadID.

La mémoire sur la carte graphique est allouée de manière analogue au C/C++, en utilisant la fonction cudaMalloc :

```
size_t size = N * sizeof(float) ;
// Allocation de la mémoire sur l'hôte
h_X = (float*)malloc(size) ;
h_Y = (float*)malloc(size) ;
// Allocation de la mémoire sur le GPU
cudaMalloc((void**)&d_X, size) ;
cudaMalloc((void**)&d_Y, size) ;
```

Les transferts de données entre le GPU et l'hôte se font avec la fonction cudaMemcpy. La fonction reçoit en argument le pointeur de destination, le pointeur du tableau source, la taille du flux de données, et enfin le sens de la transmission :

```
// Transfert du vecteur X vers le GPU
cudaMemcpy(d_X, h_X, size, cudaMemcpyHostToDevice) ;
…
// Transfert du vecteur Y vers l'hôte
cudaMemcpy(h_Y, d_Y, size, cudaMemcpyDeviceToHost) ;
```

Une fois les calculs terminés, il faut veiller à libérer la mémoire :

```
// Libération de la mémoire sur l'hôte
free(h_X) ;
free(h_Y) ;
// Liberation de la mémoire sur le GPU
cudaFree(d_X) ;
cudaFree(d_Y) ;
```

Ce premier programme CUDA est à présent complètement fonctionnel !

2.2 Utilisation de la mémoire partagée

Il est parfois nécessaire d'utiliser de la mémoire partagée lorsque plusieurs threads d'un même block ont besoin de partager des données [Nvidia 08]. Soit le problème suivant : l'interversion des éléments d'un tableau deux à deux. L'objectif primordial est de ne pas rompre la coalescence lors des accès à la mémoire globale. Le kernel CUDA correspondant est illustré à la Figure 1.

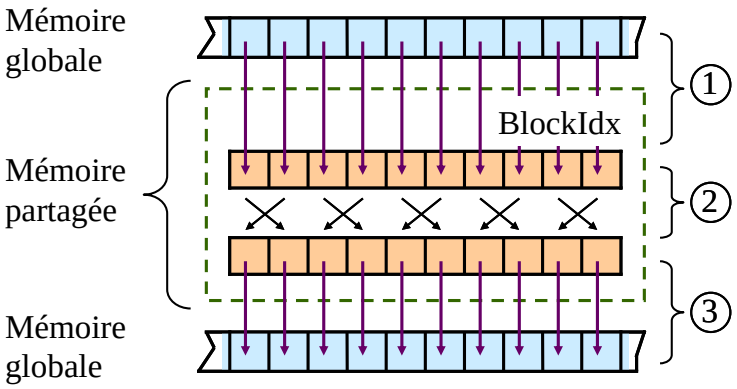


Figure 1 : Exemple de programme CUDA nécessitant l'utilisation de mémoire partagée afin de ne pas rompre la coalescence de l'accès à la mémoire globale.

Le kernel CUDA correspondant est le suivant :

```
__global__ void kernel_switch(float* X, int N)
{
    // Index dans le tableau global
    int I = blockIdx.x* (2 * BLOCK_SIZE ) + threadIdx.x ;
    int i = threadIdx.x ; // Index dans le tableau local
    __shared__ float entree[2 * BLOCK_SIZE] ;
    __shared__ float sortie[2 * BLOCK_SIZE] ;

    if (I < N / 2) // Vérification pour ne pas déborder du tableau
    {
        // Lecture coalescente du tableau global
        entree[i] = X[I] ;
        entree[i + BLOCK_SIZE] = X[I + BLOCK_SIZE] ;
        __syncthreads() ; // Synchronisation

        // Interversion des données dans la mémoire locale
        sortie[2*i] = entree[2*i+1] ;
        sortie[2*i+1] = entree[2*i] ;
        __syncthreads() ;
```

```
            // Ecriture coalescente dans le tableau global
            X[I] = sortie[i] ;
            X[I + BLOCK_SIZE] = sortie[i + BLOCK_SIZE] ;
        }
    }
```

La grille choisie est une grille à une dimension dans laquelle chaque thread effectuera une opération d'inversion. N/2 threads sont donc définis, avec N la taille du tableau à traiter. La première opération (Figure 1, 1) est l'extraction coalescente des données du tableau. Une barrière de synchronisation est placée, cette dernière est indispensable car il est nécessaire que tous les threads aient fini la lecture depuis la mémoire globale avant de passer à la suite. En d'autres termes, le tableau partagé doit être totalement initialisé avant de commencer les interversions. Les données sont ensuite interverties (Figure 1, 2), puis nouvelle synchronisation. Finalement, le résultat est retourné dans le tableau d'origine de façon coalescente (Figure 1, 3).

Cet exemple très simple illustre comment utiliser la mémoire partagée et également l'importance des synchronisations. Une dernière remarque, tout comme la mémoire partagée, les synchronisations ne s'appliquent qu'aux threads d'un même block.

3 Références

[Dowd 98] C. Severance, K. Dowd, « High Performance Computing », 2nd édition, O'Reilly Media, 1998.

[Nvidia 08] Nvidia, « Tutorial CUDA », avril 2008.

Annexe B

Calcul analytique du potentiel et du champ magnétostatique créés par un aimant permanent de forme polyédrique uniformément aimanté

Sommaire

Électrotechnique du Futur

14&15 décembre 2011, Belfort

Calcul analytique du potentiel et du champ magnétostatique créés par un aimant permanent de forme polyédrique uniformément aimanté

Christophe RUBECK, Jean-Paul YONNET, Benoît DELINCHANT et Olivier CHADEBEC

G2ELab - UMR 5269 INPG/UJF/CNRS
BP 46, 38402 Saint Martin d'Hères Cedex, France
Christophe.RUBECK@G2Elab.grenoble-inp.fr

RESUME – Ce papier présente une méthode de calcul analytique du potentiel scalaire et du champ magnétostatique créés par un aimant permanent de forme polyédrique aimanté uniformément. Le modèle coulombien permet de calculer le potentiel et le champ magnétostatique via une intégration surfacique. Cependant cette intégration sur un polygone quelconque n'est pas triviale. Une solution analytique dans le cas particulier du triangle rectangle a été développée. La méthode présentée ici est basée sur une décomposition géométrique du polygone en triangles rectangles. Les solutions associées aux triangles rectangles étant connues, il est alors possible de calculer le potentiel et le champ magnétostatique sur un aimant de forme polyédrique quelconque.

ABSTRACT – The paper presents an analytical method in order to calculate the magnetostatic scalar potential and the field created by a polyhedral shaped permanent magnet uniformly magnetized. The calculation is based on a Coulombian approach; therefore, a surface integration is needed on all the polygons that compose the polyhedron. But the integration on any polygon is not trivial. An analytical solution in the particular case of the right triangle has been developed. The method presented here is based on a geometric reduction of the polygons into right triangles. The solutions of the integrals associated with the triangles are known; thus, it is possible to calculate the magnetostatic potential and field of any polyhedral shaped magnet.

MOTS-CLES – modélisation analytique, aimant permanent, modèle coulombien, champ magnétostatique

1 Introduction

La conception de systèmes magnétiques repose principalement sur des outils de modélisation. Ces outils sont généralement basés sur une modélisation analytique en magnétostatique. Une modélisation analytique consiste à exprimer les grandeurs physiques (champ, force, etc.) sous forme de fonctions. Une modélisation numérique quant à elle demande une discrétisation géométrique du système puis la résolution des équations de la

physique sur chacun des éléments. Par conséquent, une modélisation numérique est beaucoup plus lourde (en coût de calcul et d'occupation mémoire) qu'une modélisation analytique tout en étant généralement moins précise à cause de la discrétisation. De plus, une modélisation analytique permet, si les fonctions sont dérivables, une optimisation des systèmes développés.

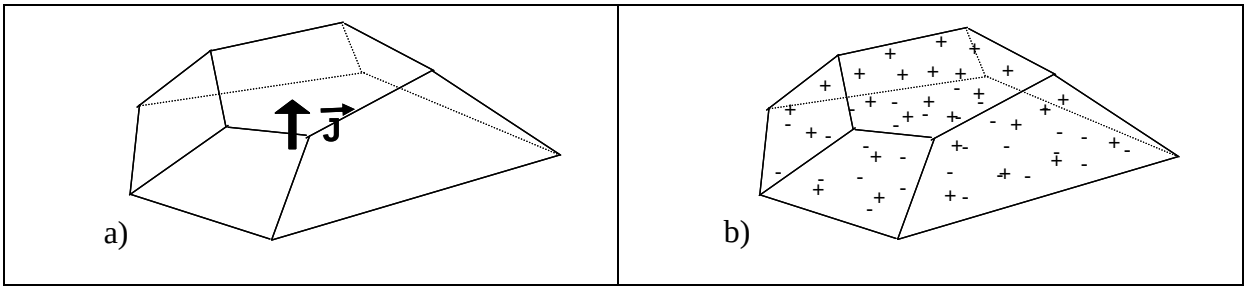


Figure 1: Exemple d'aimant de forme polyédrique. a) aimantation dans le sens de l'épaisseur de l'aimant, b) représentation coulombienne équivalente (distribution surfacique de charges magnétiques).

Cependant les modèles analytiques sont difficiles à calculer, et restent pour l'instant limités aux géométries simples. Il existe de nombreux modèles de calcul analytique de champ et de force sur des aimants permanents dans la littérature, pour une grande variété de forme d'aimants (parallélépipèdes [1-2], cylindres [3-4], anneaux [5-6], tuiles [7-8]). Une méthode de calcul analytique du potentiel et du champ magnétostatique créé par un aimant de forme polyédrique quelconque (Figure 1, a) sera présentée dans cet article.

Le calcul analytique du potentiel et du champ magnétostatique créés par un aimant permanent se fera ici par une approche coulombienne. C'est-à-dire que l'aimantation dans le volume de matière est considérée comme une distribution de charges magnétiques à la surface de ce volume (Figure 1, b).

La première étape du calcul sera donc de décomposer le polyèdre en polygones. Les charges magnétiques découlent des discontinuités de la composante normale de l'aimantation à la surface, on peut alors écrire σ_m=**J.n**, avec σ_m la densité surfacique de charge magnétostatique, **J** l'aimantation et **n** la normale à la surface de l'aimant. Le potentiel et le champ magnétique crées par chaque surface sont alors calculés de la même

manière qu'en électrostatique. Le potentiel scalaire magnétique est donc donné par l'intégrale suivante [9]:

$$\Phi(r) = \frac{\sigma_m}{4\pi} \iint_S \frac{1}{r} dS \tag{1}$$

Avec r la distance entre le point d'observation (également appelé pôle d'intégration) et la charge magnétique. On se place dans le cas d'une densité de charge surfacique uniforme (car aimantation uniforme). Le champ magnétostatique s'écrira alors [9] :

$$\vec{H}(\vec{r}) = -\vec{\nabla}\Phi(r) = \frac{\sigma_m}{4\pi} \iint_S \frac{\vec{r}}{r^3} dS \tag{2}$$

Le calcul analytique du potentiel ou du champ magnétostatique créé par une surface uniformément chargée de forme polygonale quelconque n'est pas trivial. En effet, il n'existe aucune solution générique des intégrales (1) et (2) sur des polygones quelconques. Cependant une solution analytique a été développée dans le cas particulier où le pôle d'intégration se situe sur un des sommets non rectangles d'un triangle rectangle (fig. 4). Une méthode de calcul basée sur la décomposition d'un polygone quelconque en triangles rectangles [10] sera alors présentée dans cet article.

Une intégrale surfacique sur un polygone peut s'exprimer en une somme d'intégrales sur des triangles rectangles. Les solutions analytiques des intégrales sur les triangles rectangles étant connues, les intégrales sur des polygones quelconques se calculent alors aisément, une simple analyse géométrique est nécessaire.

On exposera dans un premier temps la méthode de décomposition d'un polyèdre aimanté en polygones chargés magnétiquement grâce à une approche coulombienne. Pour permettre le calcul de (1) et (2), ces polygones sont à leur tour décomposés en triangles rectangles. Ensuite on donnera les expressions analytiques du potentiel et du champ sur les triangles rectangles, ainsi que des identités remarquables et des éléments de démonstration. On illustrera la méthode sur un aimant simple et on confrontera les résultats avec une simulation numérique par éléments finis. Puis on validera analytiquement la méthode sur un calcul de champ magnétostatique créé par un aimant parallélépipédique, on retrouvera les expressions analytiques citées par J-P. Yonnet [1].

2 Décomposition du polyèdre en polygones uniformément chargés

Afin d'illustrer la méthode, on choisit comme exemple l'aimant tétraédrique régulier présenté à la fig. 2, a). L'aimantation est uniforme selon **J**. Le modèle coulombien permet de ramener le volume aimanté à des distributions uniformes de charges sur les surfaces. De plus, grâce au principe de superposition, les contributions (en potentiel et en champ) des surfaces chargées peuvent être calculées indépendamment.

Déterminons les distributions de charges sur chacun des 4 triangles équilatéraux. Soit σ_m^+ et σ_m^- respectivement les densités de charge magnétiques correspondant aux pôles positifs et négatifs de l'aimant (fig 2, b).

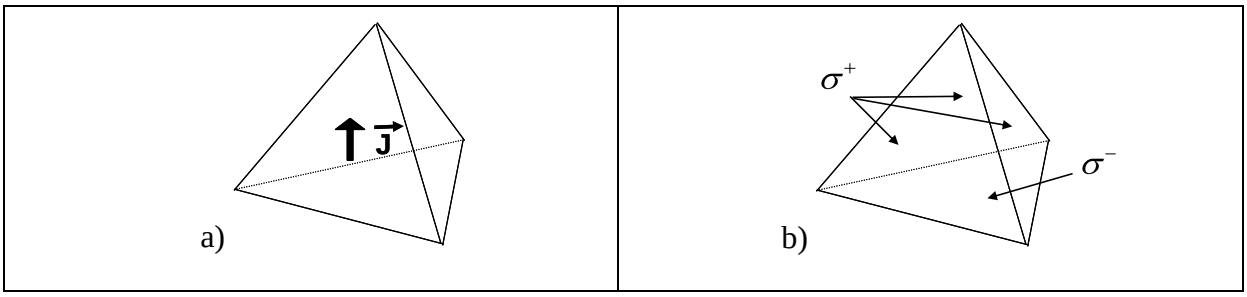


Figure 2: Aimant tétraédrique uniformément aimanté. a) aimantation selon J, b) densités de charge magnétiques du modèle coulombien équivalent.

Comme σ_m=**J.n**, la base du tétraèdre est chargée par :

$$\sigma_m^- = -\left\|\vec{J}\right\| \tag{3}$$

Les autres faces de la pyramide sont chargées par :

$$\sigma_m^+ = \left\|\vec{J}\right\|\cos\alpha = \frac{1}{3}\left\|\vec{J}\right\| \tag{4}$$

Car :

$$\alpha = \arcsin\frac{2\sqrt{2}}{3} \tag{5}$$

On peut vérifier aisément que la charge globale est conservée, en effet :

$$\sum_{i=1}^{4} \sigma_i S_i = 0 \tag{6}$$

Avec S_i la surface du i-ème triangle équilatéral.

Pour calculer le potentiel et le champ magnétostatique créés par l'aimant tétraédrique, il est maintenant nécessaire de résoudre (1) et (2) sur chacun des 4 triangles équilatéraux.

3 Décomposition d'une intégrale sur un polygone en somme d'intégrales sur des triangles rectangles

Soit un polygone de n côtés, P le pôle d'intégration et H son projeté orthogonal dans le plan du polygone. Des triangles associés aux côtés du polygone et au point H peuvent être générés [11] au nombre maximum de n. En effet il existera n-1 triangles si H se situe sur une arrête du polygone et n-2 si H se situe sur un sommet. Chacun de ces triangles est ensuite décomposé en deux triangles rectangles au maximum [12]. Une décomposition d'un polygone de n cotés génère alors au maximum 2n triangles rectangles.

Les étapes de la décomposition sont illustrées sur un triangle ABC quelconque (fig. 3) : a) Construction des triangles rectangles en L_{AB}, $AL_{AB}H$ et $BL_{AB}H$. Les solutions des intégrales associées à ces deux triangles devront être comptées négativement. En effet les surfaces que couvrent les triangles rectangles sont situées hors de ABC. b) Construction des triangles rectangles en L_{BC}, $CL_{BC}H$ et $BL_{BC}H$. La surface à intégrer étant CHB, l'intégrale associée au triangle rectangle $CL_{BC}H$ est comptée positivement et celle associée à $BL_{BC}H$ négativement. c) Construction des triangles rectangles en L_{CA}, $CL_{CA}H$ et $AL_{CA}H$. Le la même manière que précédemment, l'intégrale associée au triangle rectangle $CL_{CA}H$ est comptée positivement et celle associée à $AL_{CA}H$ négativement. d) Décomposition complète de l'intégration sur le triangle ABC. La somme de toutes les intégrales associées aux triangles rectangles, en tenant compte des signes discutés lors des étapes précédentes, est bien équivalente à l'intégrale sur le triangle ABC.

4 Calcul du potentiel et du champ magnétostatique créés par un triangle rectangle uniformément chargé

Soit le triangle rectangle uniformément chargé présenté à la Figure 4,a). Les intégrales du potentiel et des composantes du champ magnétostatique associées sont :

$$\Phi(r) = \frac{\sigma_m}{4\pi} \iint_S \frac{1}{r}\, dS = \frac{\sigma_m}{4\pi} \int_{x=0}^{x=a} \int_{y=0}^{y=bx/a} \frac{1}{\sqrt{x^2+y^2+c^2}}\, dx\, dy \tag{7}$$

$$H_x(r) = \frac{\sigma_m}{4\pi} \iint_S \frac{-x}{r^3}\, dS = \frac{\sigma_m}{4\pi} \int_{x=0}^{x=a} \int_{y=0}^{y=bx/a} \frac{-x}{[x^2+y^2+c^2]^{3/2}}\, dx\, dy \tag{8}$$

$$H_y(r) = \frac{\sigma_m}{4\pi} \iint_S \frac{-y}{r^3}\, dS = \frac{\sigma_m}{4\pi} \int_{x=0}^{x=a} \int_{y=0}^{y=bx/a} \frac{-y}{[x^2+y^2+c^2]^{3/2}}\, dx\, dy \tag{9}$$

$$H_z(r) = \frac{\sigma_m}{4\pi} \iint_S \frac{c}{r^3}\, dS = \frac{\sigma_m}{4\pi} \int_{x=0}^{x=a} \int_{y=0}^{y=bx/a} \frac{c}{[x^2+y^2+c^2]^{3/2}}\, dx\, dy \tag{10}$$

L'intégration directe de (7), (8), (9) et (10) donne les expressions analytiques du potentiel et du champ électrostatique suivantes :

$$\Phi(r) = \frac{\sigma_m}{4\pi}\left\{\frac{a}{2}\ln\left(\frac{D_{abc}+b}{D_{abc}-b}\right) - |c|\tan^{-1}\left(\frac{ab}{D_{ac}^2+|c|D_{abc}}\right)\right\} \tag{11}$$

$$H_x(r) = \frac{\sigma_m}{4\pi}\left\{\frac{-b}{2D_{ab}}\ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right) + \frac{1}{2}\ln\left(\frac{D_{abc}+b}{D_{abc}-b}\right)\right\} \tag{12}$$

$$H_y(r) = \frac{\sigma_m}{4\pi}\left\{\frac{a}{2D_{ab}}\ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right) - \frac{1}{2}\ln\left(\frac{D_{ac}+a}{D_{ac}-a}\right)\right\} \tag{13}$$

$$H_z(r) = \frac{\sigma_m}{4\pi}\left\{\tan^{-1}\left(\frac{aD_{abc}}{bc}\right) - \frac{|c|}{c}\tan^{-1}\left(\frac{a}{b}\right)\right\} \tag{14}$$

Avec : $D_{abc} = \sqrt{a^2+b^2+c^2}$, $D_{ab} = \sqrt{a^2+b^2}$, et $D_{ac} = \sqrt{a^2+c^2}$.

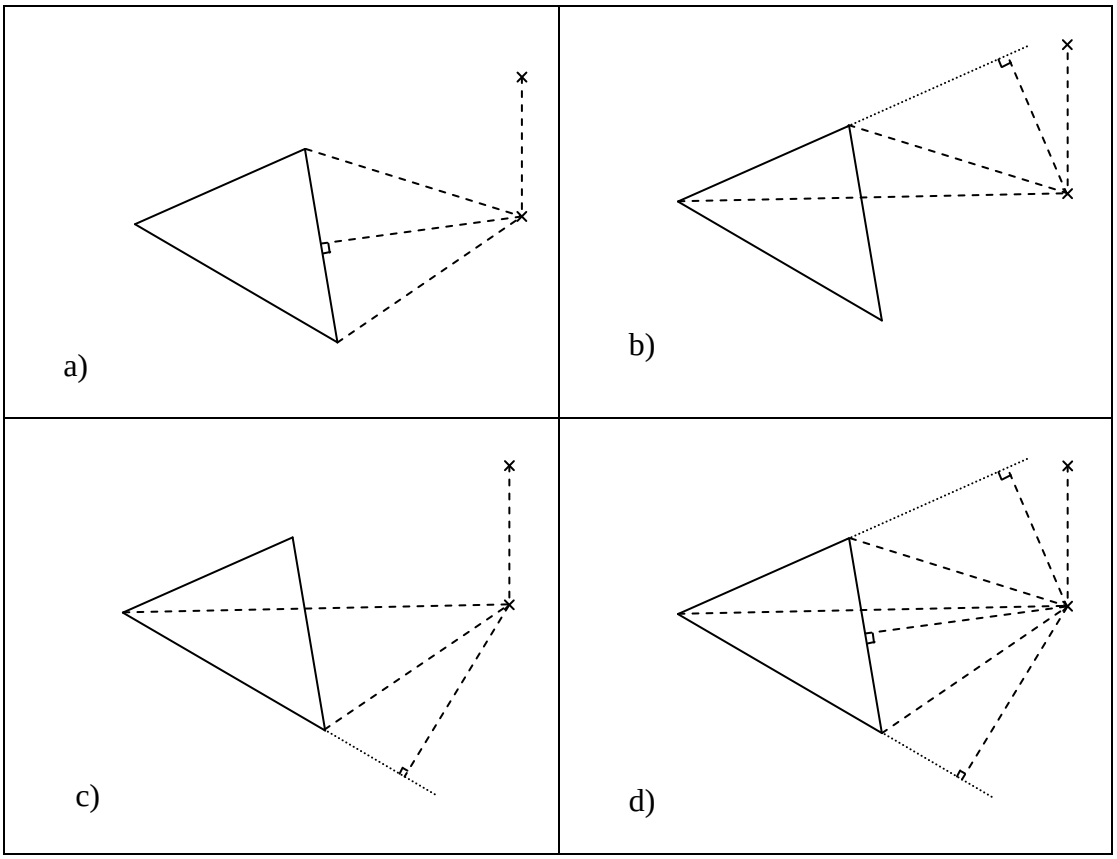


Figure 3 : Décomposition de l'intégration sur ABC depuis le pôle C en une somme d'intégrales sur des triangles rectangles.

Les solutions associées au triangle rectangle présenté à la Figure 4,b) sont :

$$\Phi(r) = \frac{\sigma_m}{4\pi}\left\{\frac{b}{2}\ln\left(\frac{D_{abc}+a}{D_{abc}-a}\right) - |c|\tan^{-1}\left(\frac{ab}{D_{bc}^2+|c|D_{abc}}\right)\right\} \tag{15}$$

$$H_x(r) = \frac{\sigma_m}{4\pi}\left\{\frac{b}{2D_{ab}}\ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right) - \frac{1}{2}\ln\left(\frac{D_{bc}+b}{D_{bc}-b}\right)\right\} \tag{16}$$

$$H_y(r) = \frac{\sigma_m}{4\pi}\left\{\frac{-a}{2D_{ab}}\ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right) + \frac{1}{2}\ln\left(\frac{D_{abc}+a}{D_{abc}-a}\right)\right\} \tag{17}$$

$$H_z(r) = \frac{\sigma_m}{4\pi}\left\{\tan^{-1}\left(\frac{bD_{abc}}{ac}\right) - \frac{|c|}{c}\tan^{-1}\left(\frac{b}{a}\right)\right\} \tag{18}$$

Avec : $D_{abc} = \sqrt{a^2+b^2+c^2}$, $D_{ab} = \sqrt{a^2+b^2}$, et $D_{bc} = \sqrt{b^2+c^2}$.

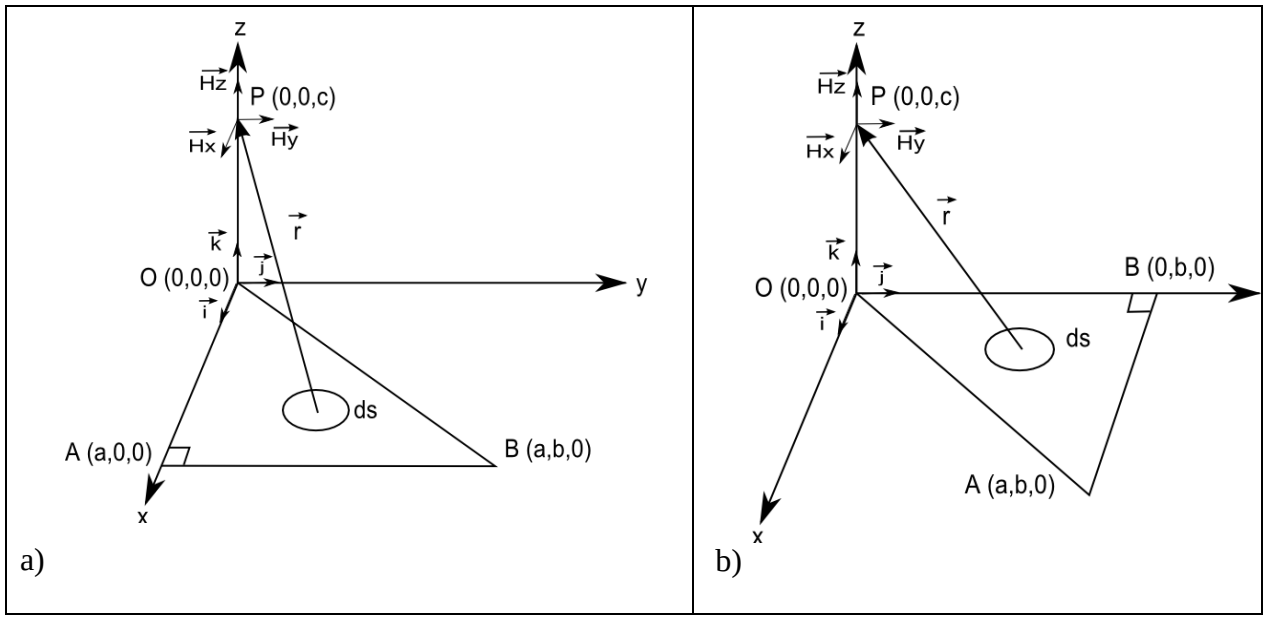


Figure 4 : Définition des triangles rectangles : a) l'angle droit se situe sur Ox, b) l'angle droit se situe sur Oy.

Remarque : la composante normale H_z du champ magnétostatique est discontinue à la traversée de la surface S. On peut raisonnablement prendre H_z=0 pour c=0 [9].

Il est maintenant possible de calculer le potentiel scalaire et le champ magnétostatique créé par un aimant de section polygonale quelconque en le décomposant en somme de triangles rectangles du type a) et b).

5 Identités remarquables et éléments de démonstration

Lors de la reconstruction de l'intégrale sur le polygone d'origine, l'assemblage des différentes solutions sur les triangles rectangles de références peut être facilitée à l'aide des identités remarquables suivantes :

$$\ln\left(\frac{D_{abc}+D_{ab}}{|c|}\right)=\frac{1}{2}\ln\left(\frac{D_{abc}+D_{ab}}{D_{abc}-D_{ab}}\right)=\ln\left(\frac{|c|}{D_{abc}-D_{ab}}\right) \tag{19}$$

$$\ln\left(\frac{D_{ac}+a}{|c|}\right)=\frac{1}{2}\ln\left(\frac{D_{ac}+a}{D_{ac}-a}\right)=\ln\left(\frac{|c|}{D_{ac}-a}\right) \tag{20}$$

$$\ln\left(\frac{D_{abc}+b}{D_{ac}}\right)=\frac{1}{2}\ln\left(\frac{D_{abc}+b}{D_{abc}-b}\right)=\ln\left(\frac{D_{ac}}{D_{abc}-b}\right) \tag{21}$$

$$\tan^{-1}\left(\frac{a\,D_{abc}}{b\,c}\right)+\tan^{-1}\left(\frac{b\,D_{abc}}{a\,c}\right)=-\tan^{-1}\left(\frac{c\,D_{abc}}{a\,b}\right) \tag{22}$$

Les formules trigonométriques suivantes sont également très utiles :

$$\tan^{-1}(x)+\tan^{-1}\left(\frac{1}{x}\right)=\frac{\pi}{2}+k\pi,\ k\ entier \tag{23}$$

$$\tan^{-1}(x)+\tan^{-1}(y)=\tan^{-1}\left(\frac{x+y}{1-x\,y}\right)+k\pi \tag{24}$$

Avec $k=0\ si\ xy<1$

$k=1\ si\ xy>0$

$k=-1\ si\ xy>1\ et\ x<0$

La démonstration des expressions données précédemment par intégration directe nécessite les étapes suivantes :

$$\int\frac{x}{k\,x+\sqrt{h^2x^2+c^2}}\left[k+\frac{h^2x}{\sqrt{h^2x^2+c^2}}\right]dx=x-2|c|\tan^{-1}\left(\frac{x}{k\,x+\sqrt{h^2x^2+c^2}+|c|}\right) \tag{25}$$

Si et seulement si $h^2=k^2+1$.

$$\int\frac{\alpha}{\sqrt{\beta\,x^2+c^2}}dx=\frac{\alpha}{\sqrt{\beta}}\ln\left(\sqrt{\beta\,x^2+c^2}+x\sqrt{\beta}\right) \tag{26}$$

$$\int\frac{\alpha\,c^2}{(x^2+c^2)\sqrt{\beta\,x^2+c^2}}dx=\frac{\alpha}{2\sqrt{\beta-1}}\ln\left(\frac{\sqrt{\beta\,x^2+c^2}+x\sqrt{\beta-1}}{\sqrt{\beta\,x^2+c^2}-x\sqrt{\beta-1}}\right) \tag{27}$$

$$\int\frac{\alpha\,x}{(x^2+c^2)\sqrt{\beta\,x^2+c^2}}dx=\frac{\alpha}{c\sqrt{\beta-1}}\tan^{-1}\left(\frac{\sqrt{\beta\,x^2+c^2}}{c\sqrt{\beta-1}}\right) \tag{28}$$

$$\int\frac{\alpha\,x^2}{(x^2+c^2)\sqrt{\beta\,x^2+c^2}}dx=\int\frac{\alpha}{\sqrt{\beta\,x^2+c^2}}dx-\int\frac{\alpha\,c^2}{(x^2+c^2)\sqrt{\beta\,x^2+c^2}}dx \tag{29}$$

Avec $\beta > 1$.

6 Application 1 : calcul du champ magnétostatique crée par un prisme aimanté

Soit le prisme à base triangulaire équilatérale présenté à la fig. 5, a), on se propose de calculer le champ normal sur un chemin perpendiculaire au prisme et le coupant en son barycentre G.

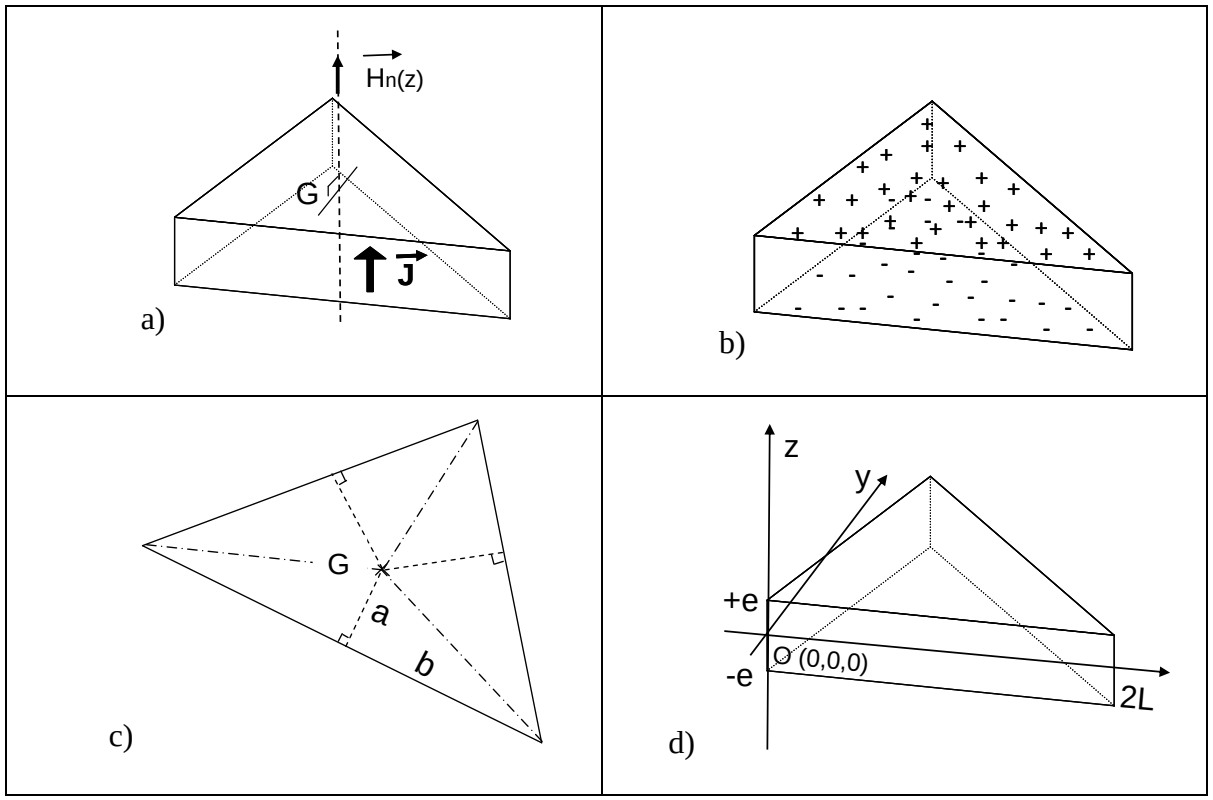


Figure 5 : Etapes du calcul du champ normal au-dessus du barycentre d'un prisme aimanté à base triangulaire équilatérale. a) Prisme aimanté à base triangulaire équilatérale, b) représentation coulombienne équivalente, c) décomposition en triangle rectangle à partir du barycentre, d) dimensions du prisme.

6.1 Calcul des densités de charges sur les faces du prisme

Le calcul des densités de charges est immédiat (fig. 5, b), on obtient $\sigma_m=\pm J$ (norme du vecteur **J**). La densité de charges est positive au-dessus (pôle +) et négative en dessous (pôle -). Les faces latérales ne sont pas chargées magnétiquement.

6.2 Décomposition des faces chargées en triangles rectangles

La décomposition en triangles rectangles est triviale, en effet on obtient sur chaque face 6 triangles rectangles identiques (voir fig. 5, c). Dans ce cas particulier du calcul du champ normal, il n'est même pas nécessaire d'effectuer une analyse pour déterminer de quels types sont les triangles rectangles (fig. 4, angle sur Ox ou Oy), ils sont équivalents ici. Pour un prisme de largeur 2L et d'épaisseur 2e (fig. 5, d) les paramètres des triangles rectangles sont :

$$a=\frac{L}{\sqrt{3}},\quad b=L,\quad c=z\pm e \tag{30}$$

Le paramètre c dépend de la face (supérieure ou inférieure) du prisme qui est considérée.

6.3 Expression analytique du champ normal

Le champ normal est la somme de 12 contributions de triangles rectangles chargés (6 pour la face du dessus et 6 pour celle du dessous). De plus, les triangles rectangles de chaque face sont identiques. Donc d'après (14) et (30) on obtient :

$$H_z(z)=\frac{6J}{4\pi}\left\{\begin{array}{l}\tan^{-1}\left(\dfrac{\sqrt{\dfrac{4}{3}L^2+(z-e)^2}}{\sqrt{3}(z-e)}\right)-\tan^{-1}\left(\dfrac{\sqrt{\dfrac{4}{3}L^2+(z+e)^2}}{\sqrt{3}(z+e)}\right)\\ +\left(\dfrac{|z+e|}{z+e}-\dfrac{|z-e|}{z-e}\right)\tan^{-1}\left(\dfrac{1}{\sqrt{3}}\right)\end{array}\right\} \tag{31}$$

6.4 Application numérique et validation

Les dimensions du prisme sont L=1mm et e=0.25mm. L'aimantation peut s'écrire :

$$\vec{J} = \frac{\vec{B}_r}{\mu_0} \tag{32}$$

Avec Br le champ rémanent qui sera pris ici à 1 Tesla, et μ_0 la perméabilité magnétique du vide. On trace à la fig. 6 le champ normal en fonction de z sur la ligne verticale passant par le barycentre. On compare les résultats obtenus avec une simulation numérique classique par éléments finis. Les deux courbes se superposent bien, les défauts observés sur le calcul numérique sont dus au maillage. On note également que le saut au niveau de la discontinuité du champ est de l'ordre de 8.10^5 A/m, ce qui correspond à la norme de **J**.

6.5 Conclusion

On a présenté un exemple simple de calcul de champ en détails. On a tout d'abord calculé les distributions charges sur les faces du prismes. Puis on a calculé le champ par une décomposition des surfaces en triangles rectangles. Les résultats ont ensuite été comparés à une simulation par éléments finis. Le calcul du champ tangentiel est plus complexe car il nécessite des projections des composantes du champ des triangles rectangles de référence dans le repère global.

7 Application 2 : Calcul du champ magnétostatique créé par un aimant parallélépipédique

On se propose en deuxième exemple de calculer le champ magnétostatique créé par un aimant parallélépipédique uniformément aimanté en utilisant la méthode de décomposition en triangles rectangles. On validera la méthode en retrouvant les formules citées par J-P. Yonnet [1].

La première étape est la décomposition géométrique du rectangle en triangles rectangles dans le plan du rectangle (fig. 7). Soit H le projeté orthogonal du pôle d'intégration dans le plan Oxy. Chaque triangle rectangle est identifié, on donne ensuite les

paramètres des triangles rectangles (Tableau 1) en fonction du type de ces derniers (angle droit sur Ox ou Oy) ainsi que le signe des intégrales.

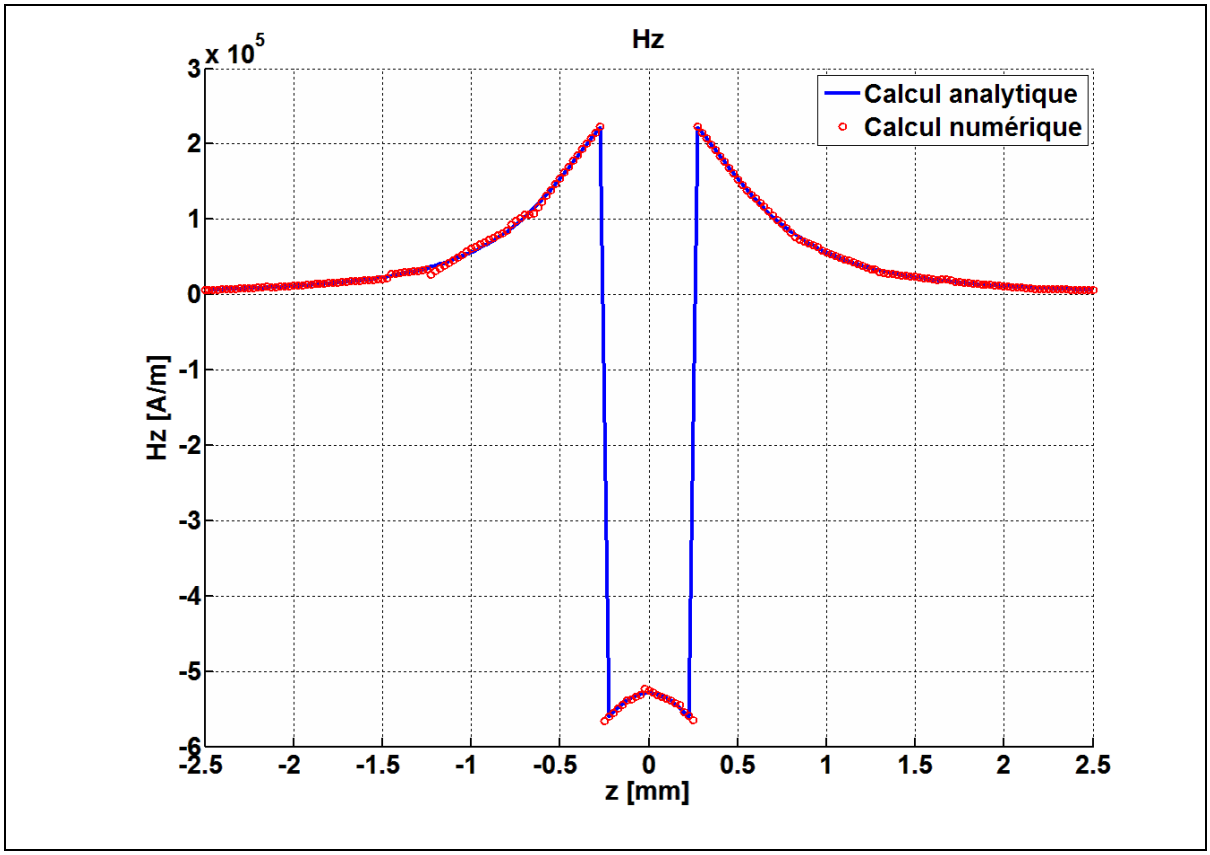


Figure 6 : normal au prisme aimanté suivant une ligne verticale passant par le barycentre.

Additionnons les contributions 1 et 5 de la composante Hz du champ magnétostatique :

$$Hz_{1+5} = \frac{\sigma_m}{4\pi}\left\{\begin{array}{l}\tan^{-1}\left(\dfrac{(x_0-m)\sqrt{(x_0-m)^2+(y_0-n)^2+z_0^2}}{(y_0-n)z_0}\right)-\dfrac{|z_0|}{z_0}\tan^{-1}\left(\dfrac{x_0-m}{y_0-n}\right)\\ +\tan^{-1}\left(\dfrac{(y_0-n)\sqrt{(x_0-m)^2+(y_0-n)^2+z_0^2}}{(z_0-m)z_0}\right)-\dfrac{|z_0|}{z_0}\tan^{-1}\left(\dfrac{y_0-n}{x_0-m}\right)\end{array}\right\} \quad (33)$$

Appliquons la formule (24) au premier et troisième terme, et la formule (25) au second et quatrième terme :

$$Hz_{1+5} = \frac{\sigma_m}{4\pi}\left\{-\tan^{-1}\left(\frac{z_0\sqrt{(x_0-m)^2+(y_0-n)^2+z_0^2}}{(x_0-m)(y_0-n)}\right)-\frac{|z_0|}{z_0}\frac{\pi}{2}\right\} \quad (34)$$

On obtient alors :

$$Hz_{1+5} = \frac{\sigma_m}{4\pi}\left\{\tan^{-1}\left(\frac{(x_0-m)(y_0-n)}{z_0\sqrt{(x_0-m)^2+(y_0-n)^2+z_0^2}}\right)\right\} \tag{35}$$

En procédant de la même manière avec les autres contributions, on obtient la composante totale du champ Hz :

$$Hz = \frac{\sigma_m}{4\pi}\left\{\begin{array}{l}\tan^{-1}\left(\frac{(x_0-m)(y_0-n)}{z_0\sqrt{(x_0-m)^2+(y_0-n)^2+z_0^2}}\right)-\tan^{-1}\left(\frac{(x_0-m)(y_0+n)}{z_0\sqrt{(x_0-m)^2+(y_0+n)^2+z_0^2}}\right)\\ -\tan^{-1}\left(\frac{(x_0+m)(y_0-n)}{z_0\sqrt{(x_0+m)^2+(y_0-n)^2+z_0^2}}\right)+\tan^{-1}\left(\frac{(x_0+m)(y_0+n)}{z_0\sqrt{(x_0+m)^2+(y_0+n)^2+z_0^2}}\right)\end{array}\right\} \tag{36}$$

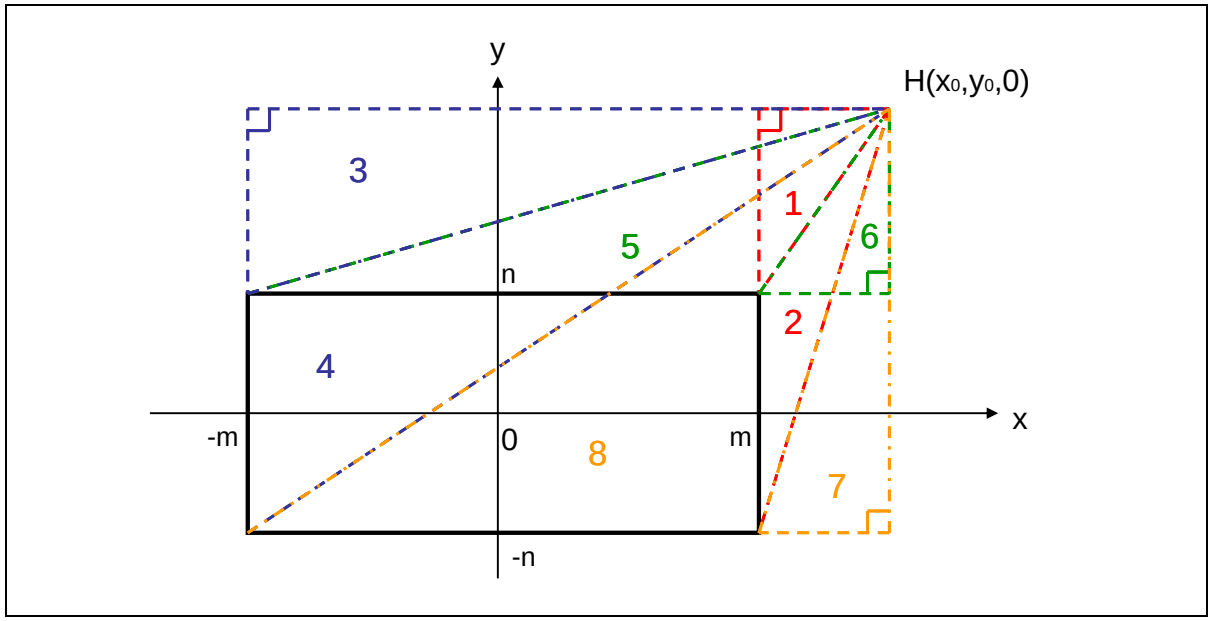


Figure 7 : Décomposition d'une intégrale sur un rectangle en somme d'intégrales sur des triangles rectangles.

On retrouve bien l'expression analytique donnée dans [1].

8 Instabilité numérique

Programmer le calcul du potentiel et du champ électrostatique créé par un polygone par la méthode de décomposition en triangles rectangles est relativement aisé. Cependant, des instabilités numériques peuvent apparaître lors de l'automatisation de la décomposition des polygones en triangles rectangles. En effet, si la projection du pôle d'intégration sur le plan du polygone se situe au voisinage d'une arrête (ou de son prolongement) ou d'un sommet du polygone, des triangles plats, c'est-à-dire dont la surface tend vers zéro, vont être générés (fig. 8). Les solutions analytiques des intégrales associées à ces triangles plats sont divergentes.

La solution est d'inclure un contrôle sur le rapport des surfaces entre le polygone et le triangle rectangle généré. Si la surface du triangle rectangle est négligeable par rapport à celle du polygone, alors il est raisonnable de considérer que sa contribution sur le potentiel ou le champ soit également négligeable, il ne sera alors pas calculé. On évite ainsi une divergence numérique sans perte notable en précision.

9 Discussion

Le calcul analytique du potentiel et du champ magnétostatique créés par une surface chargée de forme polygonale est possible à partir des travaux de Wilton [10] et Graglia [12]. Cependant, ces méthodes souffrent de quelques lacunes. En effet les démarches proposées ne sont pas aussi intuitives qu'une simple décomposition géométrique en triangles rectangles. De plus les divergences numériques sont bien mieux maîtrisées dans la méthode proposée ici, et la programmation de notre méthode s'avère également plus efficace.

10 Conclusion

Une méthode de calcul analytique du potentiel scalaire et du champ magnétostatique créés par un aimant permanent de forme polyédrique a été présentée dans cet article. Une représentation coulombienne de l'aimant permet de considérer l'aimantation de la matière comme une répartition de charges sur chaque face du polyèdre.

Tableau 1 : Paramètres des intégrales des triangles rectangles et signes des intégrales.

Angle droit sur Ox			
1 +	a=x0-m b=y0-n c=z0	2 -	a=x0-m b=y0+n c=z0
3 -	a=x0+m b=y0-n c=z0	4 +	a=x0+m b=y0+n c=z0

Angle droit sur Oy			
5 +	a=x0-m b=y0-n c=z0	6 -	a=x0+m b=y0-n c=z0
7 -	a=x0-m b=y0+n c=z0	8 +	a=x0+m b=y0+n c=z0

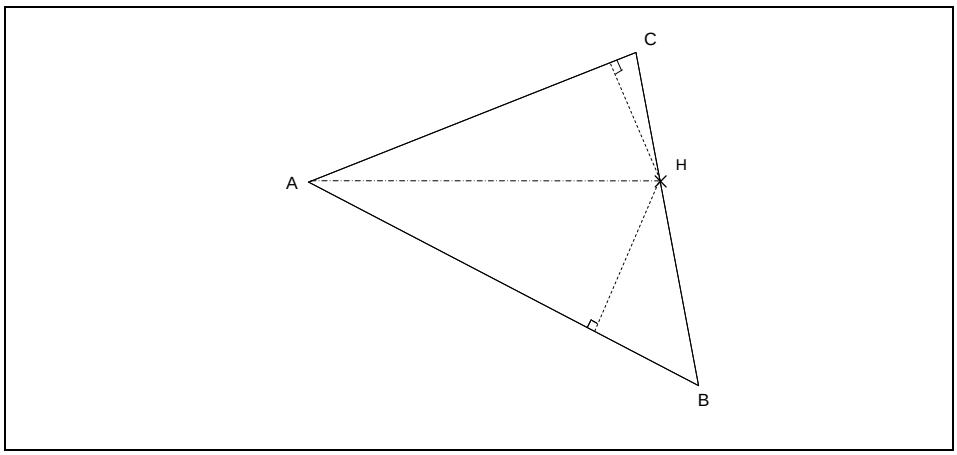


Figure 8 : Décomposition du triangle ABC avec H se situant sur une arrête. Le triangle CHB est plat.

Il est ensuite nécessaire de calculer les intégrales du potentiel et du champ sur chacune de ses surfaces. Ce calcul n'étant pas trivial, une solution consistant à décomposer géométriquement les polygones en triangles rectangles, dont les solutions analytiques des intégrales du champ et du potentiel sont connues, a été proposée dans cet article. Par conséquent, il est possible de calculer analytiquement le potentiel et le champ magnétostatique créés par un aimant de forme polyédrique quelconque.

11 Références

[1] G. Akoun and J-P. Yonnet, "3D analytical calculation of the forces exerted between two cuboidal magnets", IEEE Transactions on magnetics, vol. MAG-20, no. 5, sep. 1984.

[2] P. Elies and G. Lemarquand, Analytical optimization of the torque of a permanent-magnet coaxial synchronous coupling, IEEE Trans. Magn., vol. 34, no. 4, pp. 2267–22731, Jul. 1998.

[3] J-P. Yonnet, Permanent Magnet Bearings and Couplings, IEEE Trans. Magn., vol. MAG-17, no. 1, jan. 1981.

[4] R. Ravaud, G. Lemarquand, S. Babic, V. Lemarquand, and C. Akyel, Cylindrical Magnets and Coils: Fields, Forces, and Inductances, IEEE Trans. Magn., vol. 46, no. 9, pp. 3585, sep. 2010

[5] R. Ravaud, G. Lemarquand, V. Lemarquand, and C. Depollier, Analytical Calculation of the Magnetic Field Created by Permanent-Magnet Rings, IEEE Trans. Magn., vol. 44, NO. 8, pp. 1982-1989, aug. 2008

[6] S. I. Babic and C. Akyel, "Improvement in the analytical calculation of the magnetic field produced by permanent magnet rings", Progress In Electromagnetics Research C, vol. 5, pp. 71-72, 2008.

[7] E. P. Furlani, Formulas for the force and torque of axial couplings, IEEE Trans. Magn., vol. 29, pp. 2295–2301, Sept. 1993.

[8] H. Rakotoarison, J-P. Yonnet, and B. Delinchant, "Using coulombian approach for modelling scalar potential and magnetic field of a permanent magnet with radial polarization", IEEE Trans. Magn., vol. 43, no. 4, pp. 1261–1264, Apr. 2007.

[9] E. Durand. "Electrostatique, Tome I - Les distributions". Masson, chp. Distributions superficielles, p. 13-225, 1966.

[10] D. Wilton, S. Rao, A. Glisson, D. Schaubert, O. Al-Bundak, and C. Butler, "Potential integrals for uniform and linear source distributions on polygonal and polyhedral

domains", IEEE Transactions on Antennas and Propagation 32(3), 276–281 Mar 1984.

[11] B. Hachi-Ashtiani. "Annexe1. Calcul analytique de l'intégrale de 1/r sur un triangle". Thèse de Doctorat, Ecole Centrale Lyon, 1992.

[12] R. Graglia, "On the Numerical Integration of the Linear Shape Functions Times the 3-D Green's Function or its Gradient on a Plane Triangle", IEEE Transactions on Antennas and Propagation, vol. 41, no. 10, oct. 1993.

Annexe C

Généralités sur la transformée en ondelettes

Sommaire

1 Introduction

Les ondelettes sont une famille de fonctions déduites d'une fonction mère par des opérations de translations, dilatations et rotations. Historiquement elles sont apparues de l'analyse des signaux sismiques hautes résolutions dans le but de trouver des hydrocarbures [Bournay Bouchereau 97]. Elles ont depuis trouvées des applications dans des domaines aussi variés que le traitement du signal, le traitement d'image, la physique, etc. La transformation en ondelettes décompose un signal en une somme de petites oscillations (les ondelettes) localisées dans le temps, contrairement aux ondes stationnaires de l'analyse de fourier.

La théorie des ondelettes est très vaste, nous ne l'exposons pas de manière exhaustive. Nous nous concentrons sur les définitions et les propriétés nécessaires à l'établissement d'un algorithme de compression matricielle.

2 Transformée de Fourier

La transformée de Fourrier (TF) est une des transformations les plus utilisées en traitement du signal. Elle s'écrit pour une fonction *f* intégrable à une variable :

$$F_f(\omega) = \hat{f}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} f(t) e^{-i\omega t} dt \tag{1}$$

La transformée inverse existe et s'écrit :

$$\overline{F}_f(\omega) = f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} \hat{f}(\omega) e^{i\omega t} d\omega \tag{2}$$

La transformée de Fourier permet une analyse en fréquence de la fonction *f*. Elle consiste à projeter *f* sur des fonctions mères $e^{-i\omega t}$ qui sont localisées en fréquences mais non localisées en temps. Prenons par exemple une partition de musique qui est une suite de notes, la transformée de Fourier permet de savoir combien de do sont joués. Par contre elle ne permet pas de connaître l'instant où ils ont été joués. Nous verrons que la transformation en ondelettes est quant à elle localisée en temps et en fréquence, c'est-à-dire

qu'elle permet de savoir si un do a été joué à un tel moment et si c'était une croche, une noire ou une blanche [Bournay Bouchereau 97].

3 Définition d'une ondelette

Une ondelette doit être une fonction localisée en temps et en fréquences, ceci se traduit dans le domaine fréquentiel par la condition d'admissibilité suivante [Bournay Bouchereau 97] :

$$C_\Psi = \int_{-\infty}^{+\infty} \frac{\left|\hat{\Psi}(\omega)\right|^2}{\omega} d\omega < \infty \tag{3}$$

Où Ψ est une fonction non nulle de L^2 appelée ondelette mère (ou ondelette analysante). Toute fonction de L^2 vérifiant (3) est une ondelette. Cette condition étant très souple, il existe une grande variété d'ondelettes.

Une ondelette possède la propriété de s'annuler en *0* dans le domaine de Fourier, la condition d'admissibilité s'écrit alors pour toute fonction Ψ de L^1 :

$$\hat{\Psi}(0) = \int_{-\infty}^{+\infty} \Psi(t) dt = 0 \tag{4}$$

Cette propriété signifie qu'une ondelette doit avoir une valeur moyenne nulle dans le temps et qu'elle doit se comporter comme un filtre passe bande. Par conséquent, une ondelette est une brève oscillation. La Figure 1 présente des exemples d'ondelettes, ici le chapeau mexicain (ou ondelette Ricker) et l'ondelette de Morlet.

Une ondelette mère génère une famille d'ondelettes par des opérations de dilatation (ou de contraction) et de translation :

$$\Psi_{a,b}(t) = \frac{1}{\sqrt{a}} \Psi\left(\frac{t-b}{a}\right) \tag{5}$$

Où $a \in R^+$ et $b \in R$ sont respectivement les paramètres de dilatation (ou de contraction) et de translation. Le terme $1/\sqrt{a}$ est un coefficient de normalisation, *b* est le centre de la fonction élémentaire $\Psi_{a,b}$ et *a* sa largeur [Coulibaly 92].

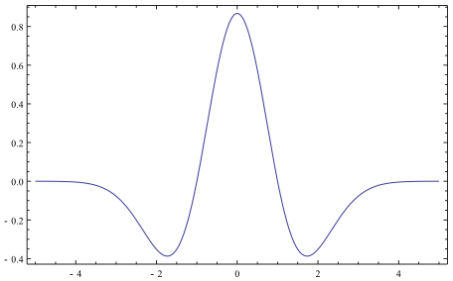
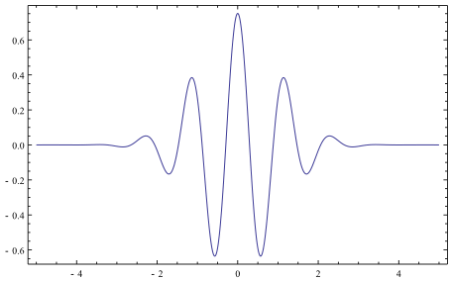
a)  b) 

Figure 1 : Exemples d'ondelettes : a) Chapeau Mexicain, b) Morlet[1].

La définition de famille d'ondelettes (5) conduit à une propriété fondamentale : si Ψ est une ondelette analysante, alors $\Psi_{a,b}$ est une base de L^2 [Bournay Bouchereau 97]. Cette propriété signifie que toute fonction de carré sommable peut être représentée en fonction d'une famille d'ondelettes. Nous pouvons alors définir la transformée en ondelettes.

4 Transformée en ondelettes continue

La transformation continue en ondelette (TOC), tout comme la transformée de Fourrier, est une transformation dans l'espace. Elle s'écrit de manière analogue à la TF, la différence réside dans le choix des fonctions de bases qui sont des familles de fonctions, ici les ondelettes, et non plus des cosinus ou des sinus. Elle est adaptée aux signaux non périodiques car elle permet une analyse temps-fréquence que ne permet pas la TF.

Elle s'écrit pour toute fonction f de L^2 :

$$W_f(a,b) = \frac{1}{\sqrt{C_\Psi}} \int_{-\infty}^{+\infty} f(t)\overline{\Psi}_{a,b}(t)dt \qquad (6)$$

$W_f(a,b)$ est appelé coefficient d'ondelette. Le facteur $1/\sqrt{C_\Psi}$ est un facteur de normalisation.

Quelques propriétés

1. Conservation de l'énergie

La transformée en ondelettes conserve l'énergie :

[1] Source : Jon Mc Loone, Wikipedia (licence Creative Commons).

$$\int |f(t)|^2 dt = \iint |W_f(a,b)|^2 da\, db \tag{7}$$

2. Inversion

La transformée en ondelettes peut s'inverser :

$$f(t) = \frac{1}{\sqrt{C_\Psi}} \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} W_f(a,b) \Psi_{a,b}(t) da\, db \tag{8}$$

Cette propriété est bien évidement essentielle dans le cadre de la compression matricielle.

3. Linéarité

$$W(\alpha f_1(x), \beta f_2(x)) = \alpha W f_1(x) + \beta W f_2(x) \tag{9}$$

Avec α et β deux réels.

4. Invariance en translation

$$W(f(x-d)) = W_f(a, b+d) \tag{10}$$

Où d est un réel.

5. Invariance en dilatation

$$W(f(\gamma x)) = \frac{1}{\gamma} W_f(\gamma a, \gamma b) \tag{11}$$

Où γ est un réel strictement positif.

5 Transformée en ondelettes discrète

Il existe plusieurs approches de discrétisation de la transformée en ondelettes. Nous verrons dans le paragraphe suivant que la plus intéressante dans le domaine de la compression des signaux utilise des familles d'ondelettes orthonormées. Toute fonction *f* de L^2 peut se décomposer en série double [Coulibaly 92] :

$$f(x) = \sum_j \sum_k w_{jk} \Psi_{jk}(x) \tag{12}$$

Avec *j* et *k* des entiers et où la famille d'ondelettes s'écrit :

$$\Psi_{jk}(x) = 2^{-j/2} \Psi(2^{-j} x - k) \tag{13}$$

Ψ_{jk} est l'ondelette dilatée d'un facteur 2^j et translatée de $2^j k$, le facteur $2^{-j/2}$ est un facteur de normalisation. Nous obtenons des coefficients d'ondelettes w_{jk} :

$$w_{jk} = 2^{-j/2} \int f(x) \Psi(2^{-j} x - k)\, dx \tag{14}$$

La transformée en ondelettes discrète d'un échantillon est sans perte d'information et elle est donc réversible exactement. De plus, les coefficients d'ondelettes sont décorrélés entre eux, le signal et le bruit ne sont pas corrélés non plus et il est alors possible de trouver des critères pour conserver uniquement le signal [Bournay Bouchereau 97].

6 Analyse multi-résolution

L'idée générale de l'analyse multi-résolution est d'observer un signal à différentes résolutions pour étudier les différences entre chaque résolution. La descente en résolution est illustrée sur la Figure 2, nous y voyons une image à différentes résolutions successives. A chaque résolution, qui est une approximation de la résolution précédente, des détails disparaissent. L'analyse multi-résolution permet de déterminer la perte d'information (ou détails) entre deux résolutions. L'objectif est de ne conserver que les détails significatifs entre chaque résolution, c'est le principe de la méthode de compression.

V_0 V_1 V_2 V_3

Figure 2 : Image vue à différentes résolutions.

Mathématiquement, une analyse multi-résolution permet d'approcher chaque fonction *f* de L^2 par une suite de fonction f_j contenant de plus en plus d'information sur *f*. L'union de

toutes les f_i contient toute l'information de f. L'intérêt est de pouvoir mesurer les changements (ou détails) entre les approximations f_j et f_{j+1}.

Soit V_j une famille de sous-espaces fermés emboîtés de L^2 dans lequel la fonction est représentée à la résolution 2^{-j} tel que [Stollnitz 95b] :

$$\cdots \subset V_{j+1} \subset V_j \subset V_{j-1} \tag{15}$$

Cette propriété signifie que toute l'information contenue à l'échelle 2^{-j} est également contenue à l'échelle $2^{-(j+1)}$. Nous écrivons également les propriétés suivantes [Bernard 99] :

$$\lim_{j \to -\infty} V_j = \bigcup_{j=-\infty}^{+\infty} V_j = L^2 \tag{16}$$

Et :

$$\lim_{j \to +\infty} V_j = \bigcap_{j=-\infty}^{+\infty} V_j = \{0\} \tag{17}$$

D'autres propriétés sont énoncées dans [Bournay Bouchereau 97]. Il est possible de définir une fonction de L^2 qui soit une base orthonormale de V_j [Gargour 09] :

$$\Phi_{jk}(x) = 2^{-j/2}\Phi(2^{-j}x - k) \tag{18}$$

Où k est un entier et où Φ_{jk} est une fonction d'échelle ou père des ondelettes. Nous définissons l'orthonormalité de la fonction d'échelle par :

$$\int_{-\infty}^{+\infty} \Phi_{jk}(x)\, \Phi_{jk'}(x)\, dx = \delta(k - k') \tag{19}$$

La fonction d'échelle a la propriété suivante [Sadiku 05] :

$$\int_{-\infty}^{+\infty} \Phi(x)\, dx = 1 \tag{20}$$

Soit W_j le supplémentaire de V_j, cet espace contient la perte d'information entre les deux approximations successives V_j et V_{j-1} :

$$V_{j-1} = W_j \oplus V_j \tag{21}$$

Cet espace peut être choisi orthogonal à l'espace V_j :

$$W_j \perp V_j \leftrightarrow \langle W_j, V_j \rangle = 0 \tag{22}$$

Il est également possible de définir une fonction de L^2 qui soit une base orthonormale de W_j :

$$\Psi_{jk}(x) = 2^{-j/2}\Psi(2^{-j}x - k) \tag{23}$$

Nous retrouvons l'ondelette définie au (13), elle constitue une base orthonormale :

$$\int_{-\infty}^{+\infty} \Psi_{jk}(x)\ \Psi_{j'k'}(x)\ dx = \delta(j - j')\ \delta(k - k') \tag{24}$$

Comme nous avons également $W_j \perp V_j$ nous pouvons écrire :

$$\int_{-\infty}^{+\infty} \Phi_{jk}(x)\ \Psi_{jk'}(x)\ dx = 0 \tag{25}$$

La base W_j contient en fait les coefficients d'ondelettes, nous pouvons alors écrire les coefficients de projection d'une fonction f de L^2 :

$$c_k^j = (f, \Phi_{jk}) = 2^{-j/2}(f(u, \Phi(2^{-j}u - k)) \tag{26}$$

$$d_k^j = (f, \Psi_{jk}) = 2^{-j/2}(f(u, \Psi(2^{-j}u - k)) \tag{27}$$

Où c_k^j est son approximation discrète dans l'espace V_j et d_k^j sa projection dans l'espace W_j.

7 Extension à la dimension 2

La construction de bases d'ondelettes à partir de l'analyse multi-résolution s'étant à la dimension 2 par le produit tensoriel suivant [Bournay Bouchereau 97] :

$$\upsilon_j = V_j \oplus V_j \tag{28}$$

Si V_j est une analyse multi-résolution de L^2, alors υ_j est une analyse multi-résolution séparable de L^2. La fonction d'échelle en 2D est alors donnée par [Waku Kouomou 93] :

$$\Phi(x,y) = \Phi(x)\Phi(y) \tag{29}$$

Le sous espace supplémentaire ω_j de υ_j dans υ_{j-1} s'écrit :

$$\omega_j = \omega_j^1 \oplus \omega_j^2 \oplus \omega_j^3 \tag{30}$$

Avec :

$$\omega_j^1 = V_j \oplus \omega_j, \quad \omega_j^2 = \omega_j \oplus V_j, \quad \omega_j^3 = \omega_j \oplus \omega_j \tag{31}$$

A chaque espace correspond une ondelette mère déterminée par :

$$\begin{aligned} \Psi^1(x,y) &= \Phi(x)\Psi(y) \\ \Psi^2(x,y) &= \Psi(x)\Phi(y) \\ \Psi^3(x,y) &= \Psi(x)\Psi(y) \end{aligned} \tag{32}$$

Les propriétés d'orthogonalités entre les ondelettes et la fonction d'échelle sont conservées en 2D.

8 Transformée en ondelettes rapide

Nous pouvons définir la transformation en ondelettes rapide (TOR) à partir de l'analyse multi-résolution [Scheiblich 11] :

$$V_{j-1} = W_j \oplus V_j = W_j \oplus (W_{j+1} \oplus V_{j+1}) = \cdots, \quad W_j \perp V_j \tag{33}$$

L'idée est toujours d'approximer une fonction *f* de L^2 à l'échelle 2^j dans le sous espace V_j, le reste de l'information (les coefficients d'ondelettes) est contenu dans le sous espace W_j. La Figure 4 illustre la transformation d'un vecteur par une TOR. L'espace V_{j-1} est l'espace normal (V_0) qui est projeté à l'étape *j* dans les sous-espaces V_j et W_j. La TOR construit les coefficients d'approximation V_j et d'ondelettes W_j uniquement pour les approximations V_{j-1}, c'est-à-dire que seuls les coefficients d'échelles sont décomposés. Il

existe une variante appelée transformée par paquets d'ondelettes qui décompose également les coefficients de détails contenus dans les bases W_j.

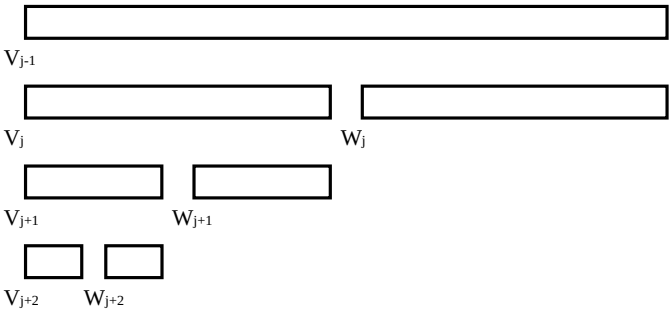


Figure 3 : Schéma de principe de la transformée en ondelettes rapide 1D.

Une conséquence importante ici est que le vecteur à transformer doit nécessairement être de dimension en puissance de deux.

La transformée en ondelettes rapide en dimension 2 repose sur le même schéma (Figure 5) [Stollnitz 95a]. Tout comme précédemment, seuls les coefficients d'approximation sont projetés dans les sous-espaces de résolutions inférieures.

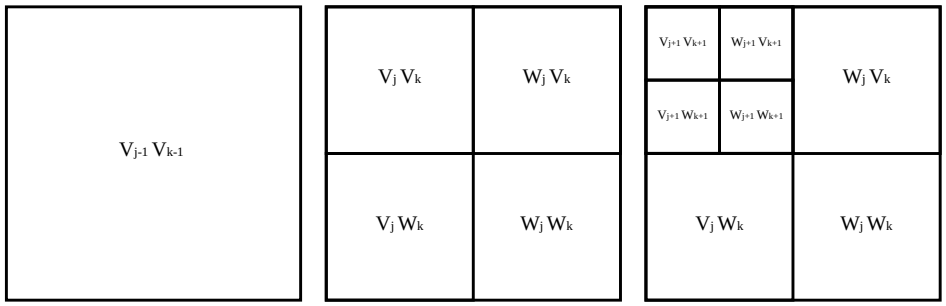


Figure 4 : Transformée en ondelettes rapide 2D.

Concrètement, grâce à la propriété de séparabilité de l'analyse multi-résolution, la transformation en 2D est effectuée en appliquant la transformation 1D sur chaque ligne et ensuite sur chaque colonne (ou inversement, l'ordre dans lequel sont effectuées ces opérations n'a pas d'importance) [Ajdari 10].

9 Quelques ondelettes

9.1 Ondelette de Haar

L'ondelette de Haar (Figure 5), d'après le mathématicien allemand Alfred Haar, est définie ainsi :

$$\Psi(t) = \begin{cases} 1, & 0 \le t < 1/2 \\ -1, & 1/2 \le t < 1 \\ 0, & \text{sinon} \end{cases} \tag{34}$$

La fonction d'échelle associée est donnée par :

$$\Phi(t) = \begin{cases} 1, & 0 \le t < 1 \\ 0, & \text{sinon} \end{cases} \tag{35}$$

C'est la première ondelette connue [Haar 1909], et c'est également l'ondelette la plus simple à comprendre et à implémenter. L'allure oscillatoire de la fonction mère est évidente, ainsi que le respect du critère d'admissibilité (moyenne nulle).

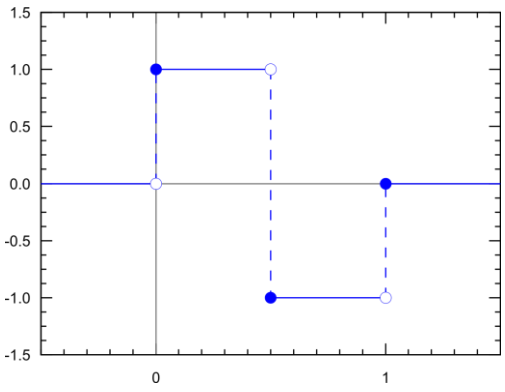


Figure 5 : Ondelette de Haar[2].

La transformation en ondelettes de Haar discrète s'écrit [Scheiblich 11] :

$$W^{Haar} = \begin{pmatrix} \Phi \\ \Psi \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{36}$$

[2] Source : Omegatron, Wikipedia (licence Creative Commons).

La transformation en ondelettes de Haar est orthogonale, sa forme orthonormale est donnée par :

$$W_{\perp}^{\ Haar} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{37}$$

Il s'agit d'une rotation de 45° de la base cartésienne suivie d'une réflexion sur le premier axe de la base.

Nous vérifions facilement ici la propriété des transformations orthogonales suivante :

$$W_{\perp} \cdot W_{\perp}^{T} = W_{\perp} \cdot W_{\perp}^{-1} = I \tag{38}$$

Nous verrons que cette propriété est très importante dans la suite car elle permet d'effectuer des produits matrice-vecteur directement sur une matrice transformée en ondelettes.

La transformation en ondelettes de Haar orthonormée peut s'écrire de manière plus générale par :

$$W_{\perp}^{\ Haar} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 & & & & & & & & \\ 1 & -1 & & & & & & & & \\ & & 1 & 1 & & & & & & \\ & & 1 & -1 & & & & & & \\ & & & & \cdots & \cdots & & & & \\ & & & & \cdots & \cdots & & & & \\ & & & & & & 1 & 1 & & \\ & & & & & & 1 & -1 & & \\ & & & & & & & & 1 & 1 \\ & & & & & & & & 1 & -1 \end{bmatrix}_{2^j \times 2^j} \tag{39}$$

Où j est un entier supérieur ou égal à 1. Nous retrouvons bien ici la contrainte sur la taille du vecteur (ou la matrice) à transformer en ondelette, c'est-à-dire d'être en puissance de deux. Remarquons également que la taille minimale du vecteur éligible à la transformation est de deux.

9.2 Ondelettes de Daubechies

Ingrid Daubechies [Bournay Bouchereau 97] a défini une famille d'ondelettes à support compact comportant un certain nombre de moments nuls : ce sont les ondelettes de Daubechies. Ces ondelettes sont définies à coefficients réels, continues et orthogonales. Les coefficients d'ondelettes sont générés en fonction du nombre de moments nuls souhaités. Une méthode est présentée par Maggy Pouliot pour calculer les coefficients des ondelettes pères en fonction du nombre *N* de zéros [Pouliot 09]. Pour *N=1*, nous obtenons deux coefficients d'ondelettes :

$$\begin{aligned} c_0 &= 1/\sqrt{2} \\ c_1 &= 1/\sqrt{2} \end{aligned} \tag{40}$$

Cette ondelette est appelée ondelette de Daubechies D2, nous reconnaissons en fait l'ondelette de Haar présenté précédemment.

Pour *N=2* nous avons l'ondelette de Daubechies D4. La Figure 6 présente l'ondelette et sa fonction d'échelle associée.

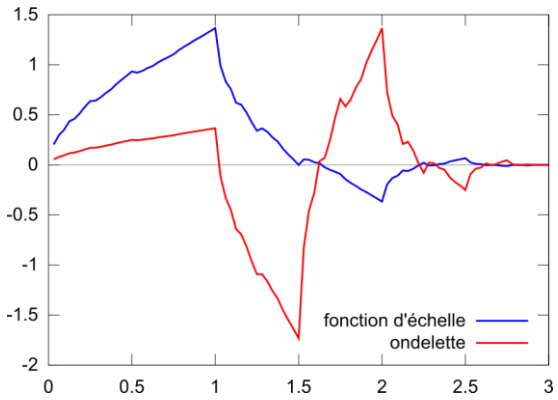


Figure 6 : Ondelette de Daubechies D4[3].

Les coefficients de l'ondelette père discrète sont [Ebrahimnejada 10] :

[3] Source : LutzL, Wikipedia (licence Creative Commons).

$$\begin{aligned} c_0 &= (1+\sqrt{3})/4\sqrt{2} \\ c_1 &= (3+\sqrt{3})/4\sqrt{2} \\ c_2 &= (3-\sqrt{3})/4\sqrt{2} \\ c_3 &= (1-\sqrt{3})/4\sqrt{2} \end{aligned} \tag{41}$$

Les coefficients de l'ondelette mère sont donnés par :

$$d_i = (-1)^i c_{3-i}, \quad i = 0,1,2,3 \tag{42}$$

La transformation peut être mise sous la forme matricielle suivante :

$$W_{\perp}^{DaubD4} = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 & & & & & & \\ d_0 & d_1 & d_2 & d_3 & & & & & & \\ & & c_0 & c_1 & c_2 & c_3 & & & & \\ & & d_0 & d_1 & d_2 & d_3 & & & & \\ & & & & \cdots & \cdots & \cdots & \cdots & & \\ & & & & \cdots & \cdots & \cdots & \cdots & & \\ & & & & & & c_0 & c_1 & c_2 & c_3 \\ & & & & & & d_0 & d_1 & d_2 & d_3 \\ c_2 & c_3 & & & & & & & c_0 & c_1 \\ d_2 & d_3 & & & & & & & d_0 & d_1 \end{bmatrix}_{2^j \times 2^j} \tag{43}$$

Où *j* est un entier supérieur ou égal à 2. Par conséquent la taille minimale du vecteur (ou de la matrice) à transformer est de quatre.

Une anecdote, la transformation en ondelette de Daubechies est utilisée dans le standard du format d'image JPEG2000 [Christopoulos 00].

10 Références

[Ajdari 10] J. Ajdari, F. Hoxha, « Parallel implementation of 2D Daubechies - D4 transform in a cluster, » Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on , vol., no., pp.377-382, 2010.

[Bernard 99] C. Bernard, « Ondelettes et Problèmes mal posés: la mesure du flot optique et l'interpolation irrégulière », Thèse de doctorat, Ecole Polytechnique, Paris, France, 1999.

[Bournay Bouchereau 97] E. Bournay Bouchereau, « Analyse d'images par transformées en ondelettes – Application aux images sismiques », Thèse de doctorat, Université Joseph Fourier – Grenoble I, Grenoble, France, 1997.

[Christopoulos 00] C. Christopoulos, A. Skodras, T. Ebrahimi, « The JPEG2000 still image coding system: an overview », Consumer Electronics, IEEE Transactions on , vol.46, no.4, pp.1103-1127, Nov 2000.

[Coulibaly 92] M. Coulibaly, « Analyse par ondelettes : quelques aspects numérique et applications à des signaux océaniques simulés et à l'estimation de densité de probabilité », Thèse de doctorat, Université Joseph Fourier – Grenoble I, Grenoble, France, 1992.

[Ebrahimnejada 10] L. Ebrahimnejada, R. Attarnejad, « Fast solution of BEM systems for elasticity problems using wavelet transforms », International Journal of Computer Mathematics, vol. 87, no. 1, pp. 77-93, 2010.

[Gargour 09] C. Gargour, M. Gabrea, V. Ramachandran, J-M. Lina, « A short introduction to wavelets and their applications », Circuits and Systems Magazine, IEEE, vol.9, no.2, pp.57-68, Second Quarter 2009.

[Haar 1909] A. Haar, « Zur Theorie der orthogonalen Funktionensysteme Erste Mitteilung », Mathematische Annalen, Volume 69, Number 3 (1910).

[Mallat 89] S.G Mallat, « A theory for multiresolution signal decomposition: the wavelet representation », Pattern Analysis and Machine Intelligence, IEEE Transactions on , vol.11, no.7, pp.674-693, Jul 1989.

[Pouliot 09] M. Pouliot, « La détermination des coefficients des ondelettes de Daubechies », Mémoire de maîtrise, Université de Laval, Québec, Canada, 2009.

[Sadiku 05] M.N.O. Sadiku, C.M. Akujuobi, R.C. Garcia, « An introduction to wavelets in electromagnetics », IEEE Microwave Magazine, pp. 63 - 72 vol. 6, no. 2, June 2005.

[Scheiblich 11] C. Scheiblich, R. Banucu, V. Reinauer, J. Albert and W. M. Rucker, « Parallel Hierarchical Block Wavelet Compression for an Optimal Compression for 3-D BEM Problems », IEEE Trans. Magn., vol. 47, no. 5, pp. 1386-1389, May 2011.

[Stollnitz 95a] E. Stollnitz, A. DeRose, D. Salesin, « Wavelets for computer graphics: a primer.1 », Computer Graphics and Applications, IEEE , vol.15, no.3, pp.76-84, May 1995.

[Stollnitz 95b] E. Stollnitz, A. DeRose, D. Salesin, « Wavelets for computer graphics: a primer.2 », Computer Graphics and Applications, IEEE , vol.15, no.4, pp.75-85, July 1995.

[Waku Kouomou 93] J. Waku Kouomou, « Ondelettes et applications en imagerie et en calcul de surfaces », Thèse de doctorat, Université Joseph Fourier – Grenoble I, Grenoble, France, 1993.

Annexe D

Couplage de la compression matricielle par ondelettes avec les matrices hiérarchiques

Sommaire

1 Introduction aux matrices hiérarchiques

Le partitionnement de la matrice d'interaction en matrices hiérarchiques est basé sur une décomposition de la géométrie par un octree. Ce partitionnement tient compte des distances entre les boites de l'octree afin de séparer proprement les interactions proches des interactions lointaines. Un bloc d'interactions dans la matrice est considéré comme champ lointain s'il satisfait à la condition d'admissibilité suivante [Wan 11] :

$$\max\{diam(B_c), diam(B_s)\} \leq \eta \cdot dist(B_c, B_s) \tag{1}$$

Où B_c et B_s sont respectivement les boites cibles et sources, et η un réel positif appelé paramètre d'admissibilité. Les termes *diam* et *dist* désignent respectivement les diamètres des boites et la distance entre les boites.

Nous illustrons le processus de partitionnement sur la Figure 1. Le problème physique est le calcul d'une distribution de charge linéique en 1D. La ligne de charges est partitionnée en plusieurs niveaux : chaque segment est divisé en deux à chaque passage de niveau. Le paramètre d'admissibilité est pris à 1, c'est-à-dire que seules les interactions d'un segment sur lui-même sont considérées en champ proche. Le niveau 0 comprend l'ensemble du domaine, il n'est donc pas admissible. Il est alors raffiné au niveau 1. La matrice d'interaction est alors partitionnée en 4 blocs, les blocs extradiagonaux sont compressibles. Le partitionnement des blocs non admissibles est obtenu en raffinant le maillage au niveau 2, et ainsi de suite.

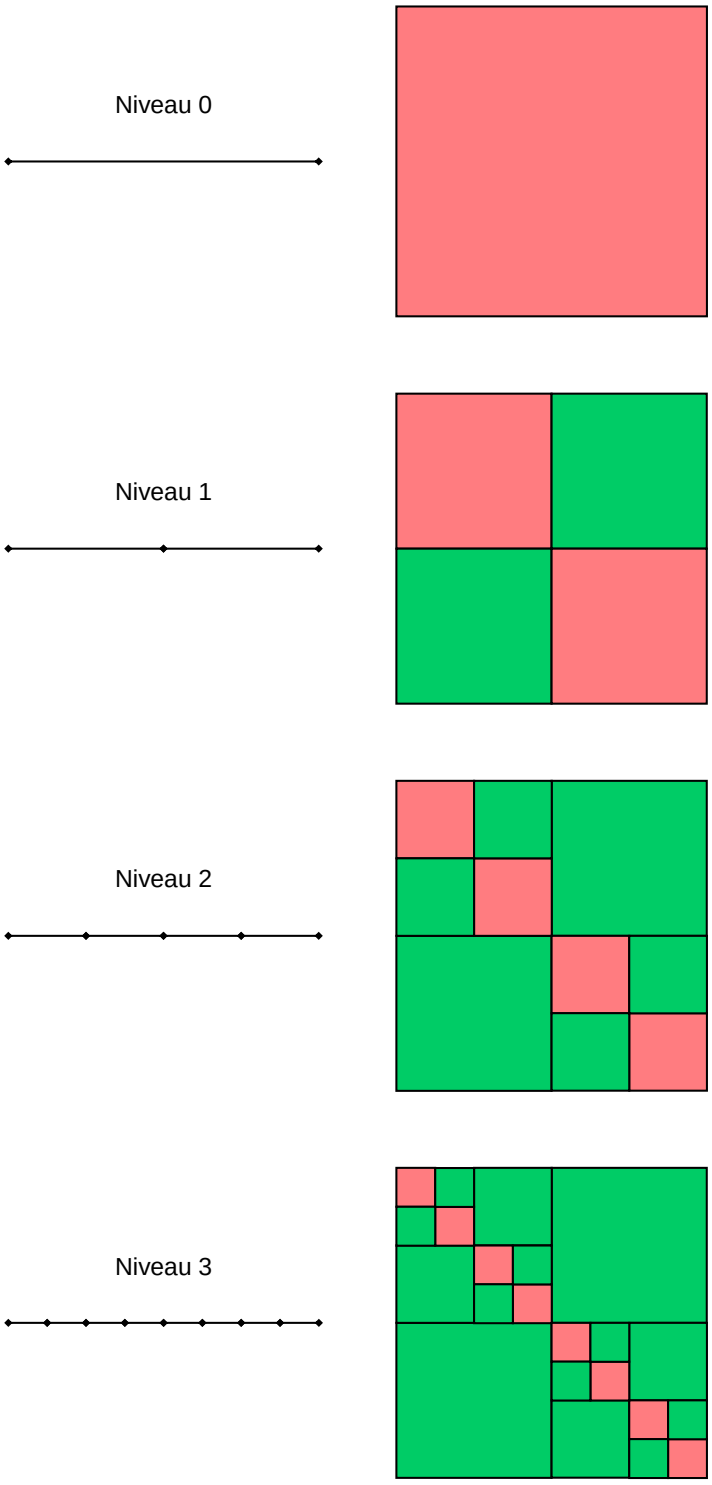


Figure 1 : Partitionnement en matrices hiérarchiques de la matrice d'intéraction. Le problème est un calcul d'une distribution de charges linéiques en 1D. Le paramètre d'admissibilité est fixé à 1.

2 Matrices hiérarchiques et compression par ondelettes

Les blocs générés par l'algorithme des matrices hiérarchiques sont de dimensions quelconques, hors pour appliquer la compression en ondelettes ils doivent être de dimensions en puissance de deux. Nous agrandissons alors les blocs, tout d'abord avec des zéros, puis en répliquant la dernière valeur de chaque ligne et de chaque colonne [McGill 92]. Les expériences suivantes sont réalisées sur le cas test par défaut et le paramètre d'admissibilité est pris égal à 2. L'octree est de niveau 4, la méthode des matrices hiérarchiques génère alors 14464 blocs pour 1,1% d'interactions non compressées.

2.1 Remplissage avec des zéros

Nous agrandissons alors les blocs avec des zéros. Nous préservons ainsi la norme de Frobenius sur laquelle nous le rappelons est basée notre critère de seuillage. Les vecteurs seront agrandis avec des zéros pour effectuer les calculs des résidus lors de la résolution itérative.

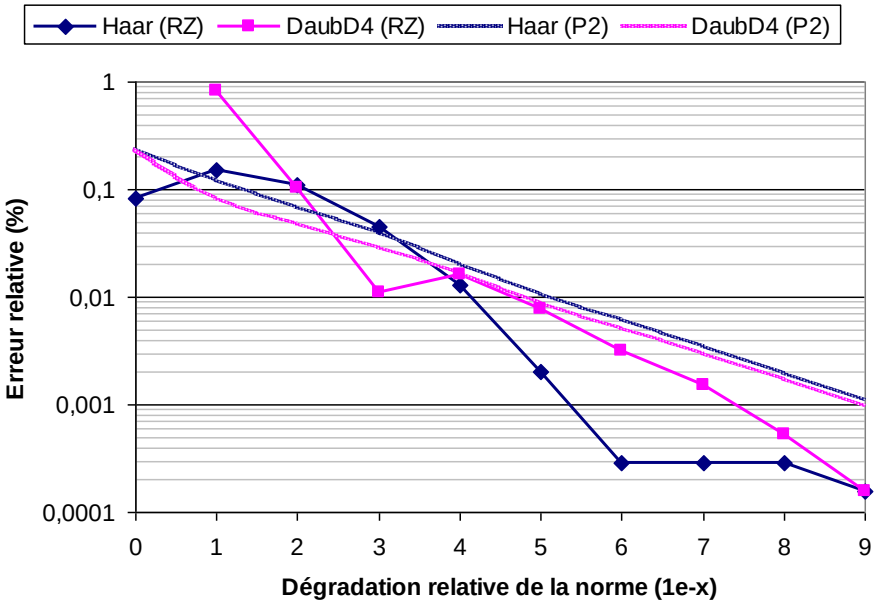


Figure 2 : Erreur relative sur la capacité en fonction de la dégradation relative de la norme des blocs agrandis avec des zéros.

Nous présentons sur la Figure 2 l'erreur relative sur la capacité en fonction de la dégradation relative de la norme des blocs. Les courbes RZ pour Remplissage Zéro sont les courbes associées à l'expérience de couplage avec des matrices hiérarchiques et les courbes P2 (Puissance 2) sont les courbes précédemment obtenues avec le partitionnement en puissance de deux. Nous voyons que la nouvelle méthode est de manière générale plus précise que celle avec le partitionnement basique en puissance de deux, cependant elle est moins stable.

La Figure 3 montre les taux de compression obtenus. Ils sont à priori moins intéressant que ceux obtenue en P2. Cependant nous jugeons l'efficacité d'une méthode sur le taux de compression pour une erreur relative donnée, c'est ce que nous voyons sur la Figure 4. Dans l'intervalle d'erreur (0,01-0,1%) qui nous intéresse le plus, le couplage avec les matrices hiérarchiques est moins performant que la méthode P2. Malgré le fait que les blocs matriciels soient très lisses en théorie, l'agrandissement avec des zéros provoque une discontinuité qui affaiblit la compression.

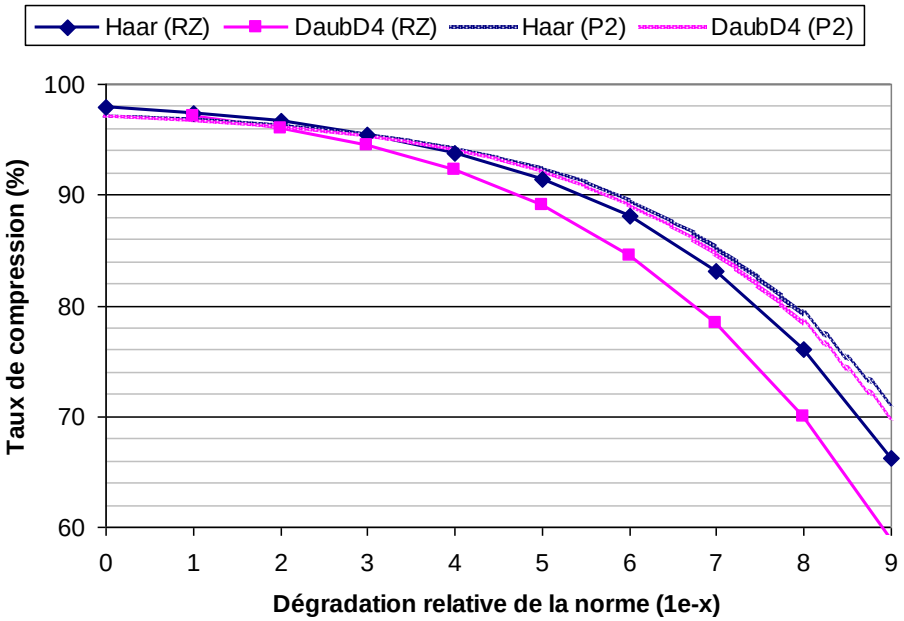


Figure 3 : Taux de compression en fonction de la dégradation relative de la norme des blocs agrandis avec des zéros.

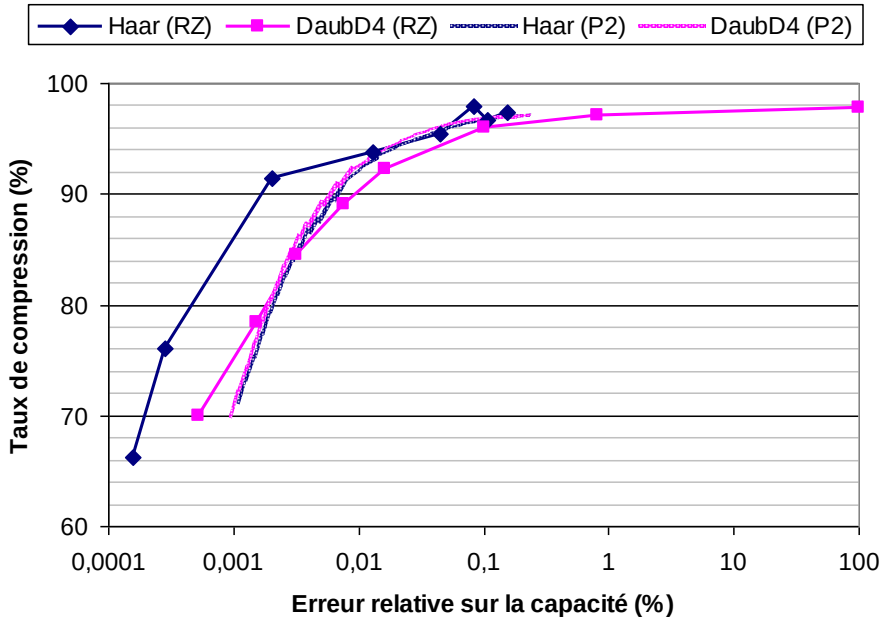


Figure 4 : Taux de compression en fonction de l'erreur relative sur la capacité.

2.2 Remplissage lisse

Nous proposons maintenant de remplir la matrice en répliquant les dernières valeurs des lignes et des colonnes. Les blocs matriciels sont alors homogènes et continus. Nous appellerons cette méthode RL pour remplissage lisse.

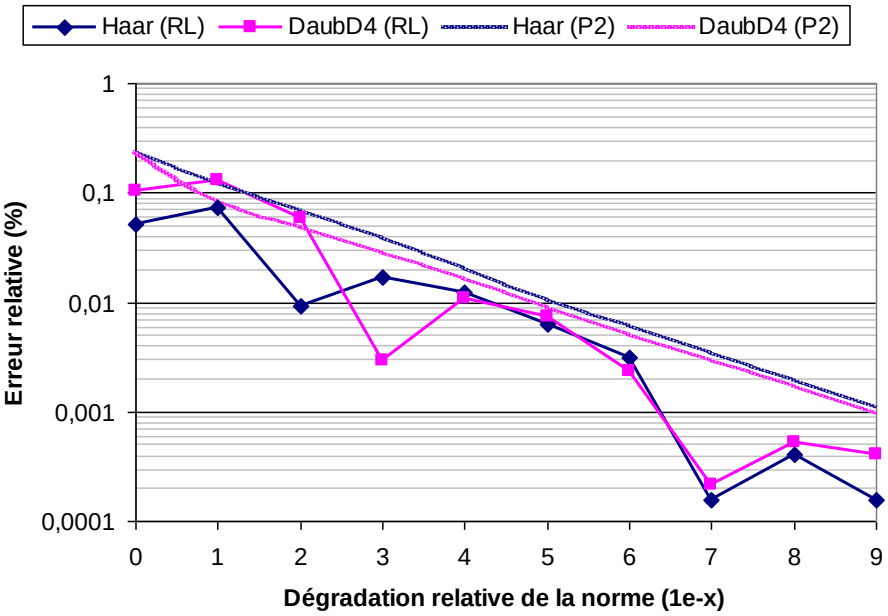


Figure 5 : Erreur relative sur la capacité en fonction de la dégradation relative de la norme des blocs agrandis en répliquant les dernières valeurs des lignes et des colonnes.

Examinons le comportement de l'erreur sur la capacité en fonction de la dégradation relative de la norme (Figure 5). Comme précédemment, les calculs sont plus précis que sur P2 mais le comportement de l'erreur n'est toujours pas très stable.

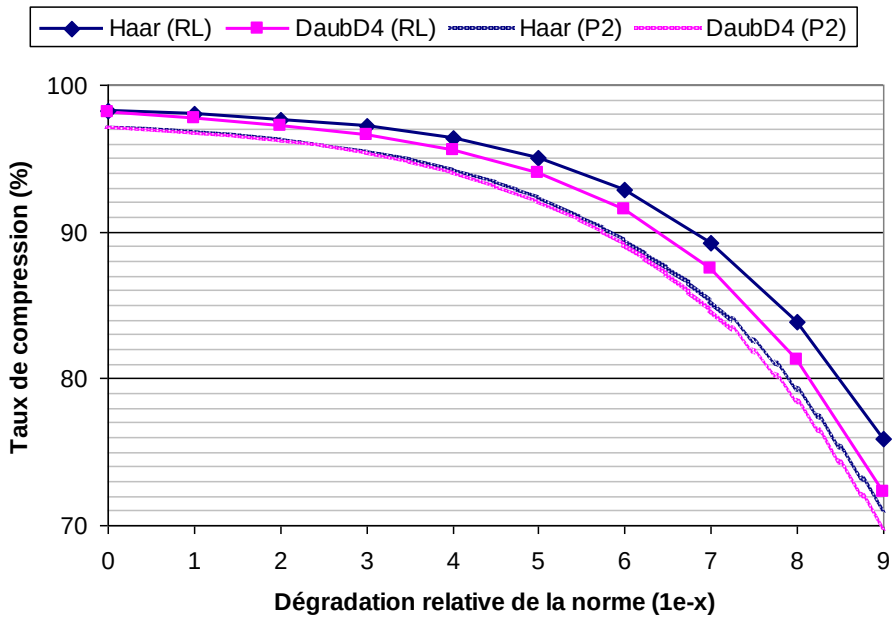


Figure 6 : Taux de compression en fonction de la dégradation relative de la norme des blocs agrandis en répliquant les dernières valeurs des lignes et des colonnes.

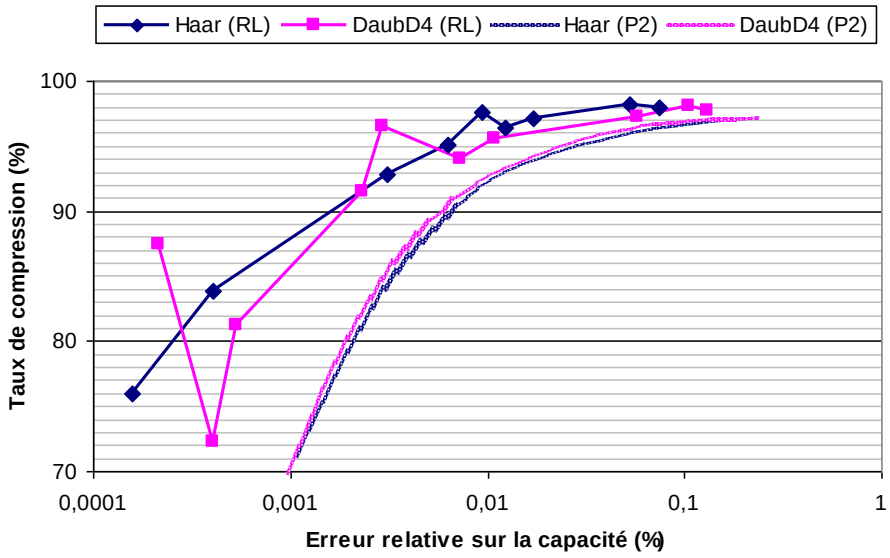


Figure 7 : Taux de compression en fonction de l'erreur relative sur la capacité.

Concernant les taux de compression (Figure 6), ils sont cette fois ci plus important que ceux obtenue avec P2. Cette méthode est donc plus précise et compresse mieux que P2. C'est ce que nous voyons sur la Figure 7, le taux de compression pour une erreur donnée est clairement meilleur que celui obtenu avec P2. Ce mode de remplissage est donc plus performant que celui avec des zéros.

3 Temps de calcul

3.1 Construction du système d'équations compressé

Nous comparons sur la Figure 8 les temps de construction du système d'équations compressé pour les méthodes RZ, RL et P2 associées à l'ondelette de Haar. Nous avons porté la méthode RZ sur GPGPU et la comparons avec la méthode P2 sur GPGPU. La méthode RL n'a pas été portée car elle est complexe à paralléliser. Nous observons sur les temps CPU que la manière de remplir les blocs agrandis artificiellement n'a pas d'incidence sur les temps de calcul et malgré le fait que nous ayons une importante quantité de blocs (15000-30000) les temps d'intégration sont à peine plus élevés que pour P2 (<1% pour 30.000 éléments). Les performances du GPGPU ne sont pas aussi intéressante en RZ qu'en P2 : nous notons une accélération d'environ 3 pour 25.000 éléments et 5 pour 30.000. Nous remarquons une petite « bosse » sur la courbe RZ GPGPU, elle est provoquée par un nombre important de blocs qui sont trop petits pour être traités sur GPU. La méthode RZ sur GPGPU est entre 2 et 5 fois plus lente que P2 sur GPGPU à cause de la grande quantité de blocs. Cette grande quantité de blocs est pénalisante car elle occasionne un grand nombre d'appels aux fonctions CUDA et de plus pour des travaux qui n'exploitent pas pleinement les capacités de calcul du GPU.

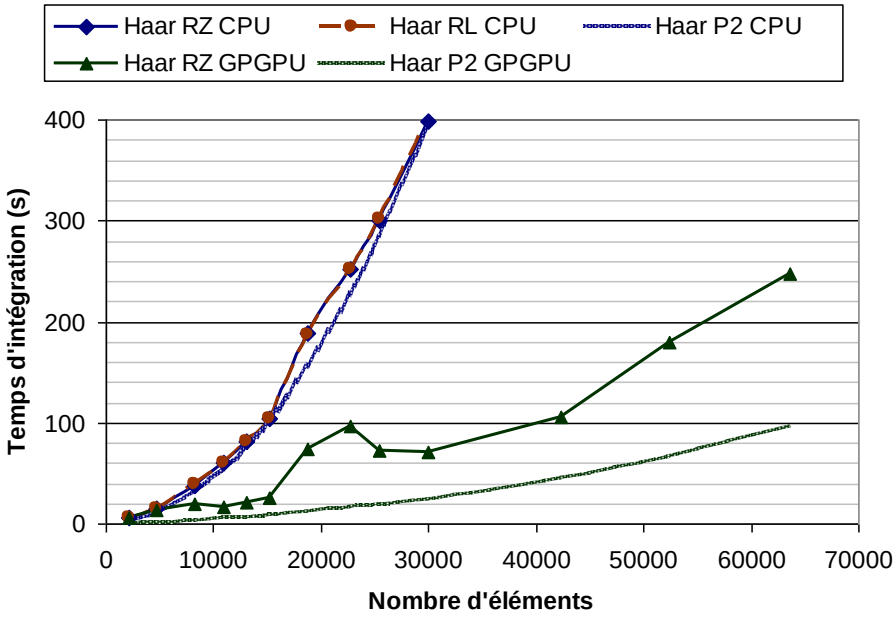


Figure 8 : Temps d'intégration et de compression en fonction du nombre d'éléments pour différentes méthodes de compression sur architectures CPU et GPGPU.

3.2 Résolution du système d'équations

Nous présentons les temps de résolution sur la Figure 9. Les temps CPU de la méthode RL sont légèrement meilleurs (8% à 30.000 éléments) que RZ car le taux de compression est meilleur, il y a donc moins d'opérations à effectuer lors des calculs des résidus. Les temps de la méthode RZ sur GPGPU sont plus élevés que pour le CPU, nous avions déjà fait ce constat sur la méthode P2. Ici l'écart est plus important encore du au grand nombre de blocs. De manière générale, la résolution pour les méthodes RZ et RL est bien moins performante que pour P2, surtout sur GPGPU où elle est de 3 à 6 fois plus lente.

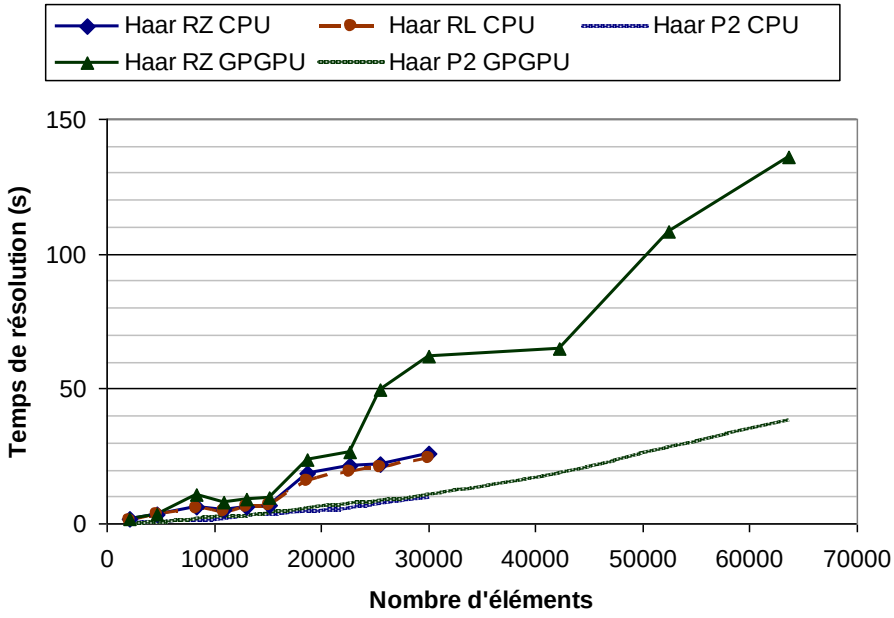


Figure 9 : Temps de résolution en fonction du nombre d'éléments pour différentes méthodes de compression sur architectures CPU et GPGPU.

4 Conclusion

Nous avons couplé la méthode de compression matricielle avec les matrices hiérarchiques. Ces dernières partitionnent la matrice d'interaction en blocs très homogènes via une analyse de la géométrie du maillage. Cependant ces blocs sont de dimensions quelconques, nous les avons alors agrandis à la puissance de deux supérieure. Nous comblons le vide dans les blocs tout d'abords avec des zéros, mais celle solution introduit une discontinuité qui nuit à la compression, puis en répliquant les dernières valeurs des lignes et des colonnes, ainsi le bloc reste homogène. Nous obtenons un meilleur rapport entre précision et taux de compression que celui obtenu avec le partitionnement basique en puissance de deux. Cependant cette méthode génère un très grand nombre de blocs ce qui nuit au parallélisme. De plus le critère de seuillage n'est pas optimal car il ne prend pas en compte l'agrandissement artificiel des blocs.

Pour conclure sur le partitionnement à adopter dans le cadre de la compression matricielle par ondelettes, il est plus intéressant d'utiliser le couplage avec les matrices hiérarchiques sur CPU car le taux de compression est meilleur pour un coup calculatoire

légèrement supérieur. Par contre sur architecture GPGPU c'est le partitionnement basique en blocs de puissance de deux qu'il est préférable d'utiliser.

5 Références

[McGill 92] K. McGill, C. Taswell, « Length-preserving wavelet transform algorithms for zero-padded and linearlyextended signals », Tech. Rep., Veterans Affairs Medical Center, Palo Alto, CA, 1992.

[Wan 11] T. Wan, Z. N. Jiang, Y. J. Sheng, « Hierarchical Matrix Techniques Based on Matrix Decomposition Algorithm for the Fast Analysis of Planar Layered Structures », Antennas and Propagation, IEEE Transactions on, vol.59, no.11, pp.4132-4141, Nov. 2011.

Calcul hautes performances pour les formulations intégrales en électromagnétisme basses fréquences - Intégration, compression matricielle par ondelettes et résolution sur architecture GPGPU

Résumé : Les méthodes intégrales sont des méthodes particulièrement bien adaptées à la modélisation des systèmes électromagnétiques car contrairement aux méthodes par éléments finis elles ne nécessitent pas le maillage des matériaux inactifs tel que l'air. Ces modèles sont donc légers en termes de nombre de degrés de liberté. Cependant ceux sont des méthodes à interactions totales qui génèrent des matrices de systèmes d'équations pleines. Ces matrices sont longues à calculer en temps processeur et coûteuses à stocker dans la mémoire vive de l'ordinateur. Nous réduisons dans ces travaux les temps de calcul grâce au parallélisme, c'est-à-dire l'utilisation de plusieurs processeurs, notamment sur cartes graphiques (GPGPU). Nous réduisons également le coût du stockage mémoire via de la compression matricielle par ondelettes (il s'agit d'un algorithme proche de la compression d'images). C'est une compression par pertes, nous avons ainsi développé un critère pour contrôler l'erreur introduite par la compression. Les méthodes développées sont appliquées sur une formulation électrostatique de calcul de capacités, mais elles sont à priori également applicables à d'autres formulations.

Mots clefs : Calcul hautes performances, méthodes intégrales, compression matricielle par ondelettes, architecture GPGPU

High performance computing for integral formulations in low frequencies electromagnetism – Integration, wavelets matrix compression and solving on GPGPU architecture

Abstract : Integral equation methods are widely used in electromagnetism modeling because, in opposition to finite element methods, they do not require the meshing of non-active materials like air. Therefore they lead to formulations with small degrees of freedom. However, they also lead to fully dense systems of equations. Computation times are expensive and the storage of the matrix is very expensive. This work presents different parallel computation strategies in order to speed up the computation time, in particular the use of graphical processing units (GPGPU) is focused. The next point is to reduce the memory requirements thanks to wavelets compression (it is an algorithm similar to image compression). The compression technique introduces errors, therefore a control criterion is proposed. The methodology is applied to an electrostatic formulation but it is general and it could also be used with others integral formulations.

Keywords : High performance computing, integral methods, wavelets matrix compression, GPGPU architecture